REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	18 April 1990	Final

4. TITLE AND SUBTITLE
Optimization of a Transient Power Stability Program on a Vector Supercomputer; Theory and Application

5. FUNDING NUMBERS
N/A

6. AUTHOR(S)
Gregg F. Degen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
HQDA, MILPERCEN (DAPC-OPB-D)
200 Stovall St
Alexandria, VA.

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release
Distribution unlimited

12b. DISTRIBUTION CODE
N/A

13. ABSTRACT (Maximum 200 words)

Detailed simulations of large interconnected power distribution systems currently cannot be performed. Vector supercomputers have the capability to solve the great number of differential and non-linear simul taneaous equations associated with the models, but programs have not been developed for them. This thesis investigates the optimization of a current power stability program installed on a Cray Y-MP vector supercomputer. Theory and models for power system stability are reviewed and an outline of the program is given. A discussion of vector supercomputer pipelining and the Cray Y-MP architecture are included. Programming constructs in FORTRAN which inhibit compiler optimization and vectorization are discussed, and program modification techniques to overcome these are presented. The modifications made to the power stability program are discussed and their effects on program execution time and output are presented. It is shown that significant speed up in program execution time can be achieved with no change in program output.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Cray Y-MP, vector supercomputer, optimization, vectorization, power stability,	131
	16. PRICE CODE: N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

OPTIMIZATION OF A POWER TRANSIENT STABILITY PROGRAM ON A VECTOR SUPERCOMPUTER, THEORY AND APPLICATIONS

by

Gregg Francis Degen

OPTIMIZATION OF A POWER TRANSIENT STABILITY PROGRAM ON A VECTOR SUPERCOMPUTER, THEORY AND APPLICATIONS

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Electrical Engineering

---

by

Gregg Francis Degen

Spring 1990

OPTIMIZATION OF A POWER TRANSIENT STABILITY PROGRAM ON A VECTOR SUPERCOMPUTER, THEORY AND APPLICATIONS

A Thesis

Presented to the

Faculty of

San Diego State University

---

by

Gregg Francis Degen

Spring 1990

Approved by:

[illegible] April/18/90
Date

Kris Stewart

[illegible]

ACKNOWLEDGEMENTS

I would like to thank the members of my thesis committee, Dr. Ramon Betancourt, Dr. Kris Stewart, and Dr. Alexander Iosupovici, for their time and effort. I also wish to express my appreciation to Susan Archer and Steve Behling for their help using and programming the Cray Y-MP computer. Most of all I thank my wife Christa for enduring the long hours required to complete this work.

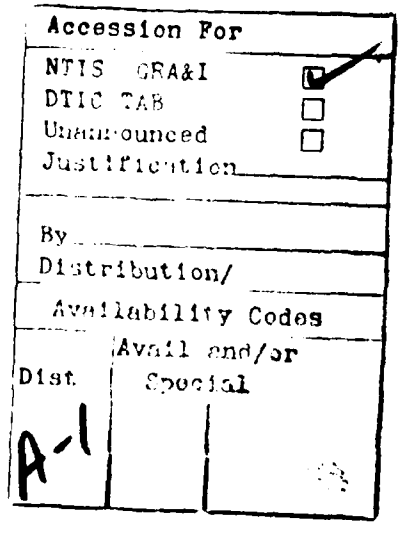


TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Over the years large interconnected power systems have developed due to the great distances between power generation plants and the loads they service. Daily load fluctuations have contributed to the interconnection of systems as power companies exchange power to efficiently utilize generation plants. These interconnections introduce external influences to a company's power system. These influences require analysis of large electrical networks to properly design a system and to properly control the flow of power in the system. The interconnections simultaneously helped and threatened system stability. The interties supply external power which helps stabilize a company's system during internal disturbances. The interties can also introduce large unexpected demands for power during disturbances external to the company boundaries.

Often, a utility company's computers do not have the capacity and power required to solve, in detail, the large networks required to simulate these conditions.

Therefore the amount of detail included in a power stability study is frequently limited.

Fourth and fifth generation computers developed during the last decade can provide the high speed computations required to solve this type of large network problem. According to Betancourt and Gomez (3), due to the declining cost of processors, there is great interest in applying parallel processing to power system analysis in the hopes of obtaining real time solutions. To fully utilize these computers old programs must be modified or new programs must be developed. They state that currently the most rapid advances in this area can be made by adapting a specific algorithm to utilize the computer's speed. The installation of the Western States Coordinating Council's (WSCC) power stability program on a Cray X-MP-48 is sighted as an example. With only compiler optimization, the program's performance was superior compared to other computer systems (3). Modification of the program, so that it more fully utilizes the computer's capabilities, promises to further reduce turn-around time and cost.

Optimizing a power stability program will not only solve the network quicker, but will allow system designers to better plan system expansion and upgrade, and will help system operators more efficiently utilize

existing resources. Designers could include more detailed models in the program better simulating actual system response to disturbances. System operators, with detailed system modeling, could more efficiently control the flow of power in the existing system. This will reduce the need to install additional transmission lines while still providing the required power. According to Betancourt and Gomez (3), the problem of optimizing a program to achieve the high computational speed possible on vector supercomputers greatly depends on subtle programming techniques.

Optimization of the WSCC's power stability program on the Cray Y-MP vector supercomputer was investigated to measure the speed up attained through compiler optimization and vectorization and code modification. Chapter II discusses power stability theory and system models. The number of simultaneous and differential equations associated with the models and network are introduced along with a system solution algorithm. Chapter III discusses the WSCC's Power Stability Program and outlines its solution techniques. Chapter IV provides an overview of computer classifications and pipelined computers. Chapter V presents details of the Cray Y-MP vector supercomputer. Chapter VI presents techniques to optimize and vectorize FORTRAN on vector

supercomputers. Chapter VII details the code modifications made to the stability program and the speed up attained.

CHAPTER II

POWER SYSTEM STABILITY THEORY

Introduction

Stevenson (17) defines power stability as the property of a power system which enables the synchronous machines in the system to return to a normal operating state following a disturbance. The purpose for power stability studies, as described by Gonen (8), is to find the maximum and minimum fault currents and voltages at all points in the system for different types of faults. With these values, system operators can develop protection schemes and select relays and circuit breakers so the system can be saved in minimum time and with minimum disturbance. These plans are then tested by performing additional stability studies.

Types of Stability Studies

Three types of power stability studies are conducted steady state, dynamic, and transient.

Steady State Studies

Steady state stability studies, also called load-flow studies, examine the system during normal operating conditions. These conditions include normal daily power fluctuations and the removal of transmission lines and machines from service for routine maintenance. According to Glover (7), steady state studies check that phase angles across transmission lines and bus voltages are within established tolerances, and that generators, transmission lines, transformers, and other equipment are not overloaded.

Dynamic Studies

Dynamic stability studies examine the system several minutes after a major disturbance or fault. Glover (7) states that system changes resulting from the disturbance or fault may actually be destabilizing, even though the original fault was not. The actions of turbine-governors, excitation control systems, tap-changing transformers and power system dispatch centers do not immediately effect the system. These delayed actions can effect the system similar to a disturbance or fault. Dynamic studies allow time for system changes to take place and indicate whether any part of the system was

overloaded or if the system will continue to operate safely.

Transient Studies

Transient stability studies examine the system immediately following a major disturbance or fault. Glover (7) states that their purpose is to determine whether the machines in the system will return to synchronous frequency (60Hz) with acceptable power angles, bus voltages, and power flow. According to Stevenson (17), the most critical disturbances are transmission system failures, sudden load changes, loss of generating units, and line switching. Transient studies are critical for system planning and operation. If the system is going to fail it will most likely do so immediately after a disturbance. Stevenson (17) states that to perform a transient study the network state before the fault, during the fault, and after the fault is cleared must be known.

The next sections describe different types of faults and some of their effects, component models and their equations, and a general solution method.

Types of Faults and Their Effects

Gonen (8) classifies faults and disturbances into three major categories shunt, series and simultaneous. Shunt faults are the most common and are considered the most serious. Series faults are the second most common, but are less serious than shunt faults. Simultaneous faults are very rare and not much work has been done on them.

Shunt Faults (Short Circuits)

Shunt faults occur when one or more phases of the system are shorted either directly or through a small resistance to ground. Shunt faults are divided into two categories balanced, or symmetrical three phase faults, and unbalanced, or unsymmetrical faults.

Balanced three phase faults. During a balanced fault all phases are simultaneously grounded. They normally produce the largest fault currents and the greatest change in the flux-leakages in synchronous machines. They are studied to size circuit breakers and set relay values. Balanced faults are rare, but due to the high currents they produce, they are potentially the most damaging to the system. High currents are produced

since the generators contirue to provide current and the motors' inertia cause them to act as generators providing additional current (8).

Unbalanced faults. During an unbalanced fault one or two phases short either to each other or to ground. The system reacts less radically than during a balanced fault, since power is still delivered through the ungrounded phases. Unbalanced faults include single-line to ground, double-line to ground, and line-to-line shorts. Analysis of unbalanced faults requires the use of symmetrical components. Models and equations for unbalanced faults using symmetrical components can be found in references (2;8;17).

Single-line to ground faults are the most common and occur when one phase of the system is shorted to ground. According to Gonen (8), they can produce a larger fault current than balanced three phase faults under two conditions, if the generators have low impedances or solid grounds and, if the fault is on the wye side of a delta-wye transformer.

Line-to-line faults are the second most common type of fault and occur when two phases of the system short together. The currents caused by these faults are usually not critical to system stability.

Double-line to ground faults are the third most common and occur when two phases of the system are simultaneously shorted to ground. The currents caused by these faults are usually not critical to system stability.

Series Faults (Open Conductors)

Series faults occur when impedances in the three phases become unbalanced. There are three types of series faults one-line open, two-lines open, and unbalanced series impedance conditions. All three types are easily solved using symmetrical components and do not cause critical currents in the system. Gonen (8) provides diagrams and equations to calculate the fault currents and voltages for each phase.

Simultaneous Faults (Two or More Faults)

Gonen (8) identifies four categories of simultaneous faults: a shunt fault at two different points in the system, a series fault at two different points in the system, a shunt fault at one point and a series fault at another, and a series fault at one point and a shunt fault at another. Simultaneous faults are very rare since they must occur at two distant points within one to two cycles of each other. Since most faults are cleared

within the first few cycles simultaneous faults can usually be analyzed as two separate disturbances

Power System Modeling

Models for several power system components are presented next to demonstrate the level of sophistication and complexity associated with modeling a power system. All possible models are not covered, for a more complete and detailed presentation, refer to (1;2;4).

Transmission Line Models

Modern power distribution networks increasingly contain both alternating current (A.C.) and direct current (D.C.) transmission lines. The model chosen to represent a transmission line depends on the line's type and length. General models for A.C. and D.C. lines are presented in this section.

Alternating Current line models. According to Stevenson (17), A.C. transmission line models are divided into three categories short, medium, and long. Short lines have lengths of fifty miles or less, medium lines have lengths of fifty to one hundred and fifty miles, and long lines are over one hundred and fifty miles long.

Since line lengths and types are known when studies are conducted, actual values for resistance, capacitance, and inductance can be calculated and used in the stability program.

Short transmission lines are modeled by placing the values of resistance and inductance in series between two buses. The capacitance is usually small and therefore ignored. If the capacitance is to be represented then the medium length line model is used.

Medium transmission lines are modeled using either the nominal-π or nominal-T equivalent circuits. The nominal-π equivalent circuit is the most common model. It places the resistance and inductance in series between two buses and places half the capacitance value in shunt at each bus.

Long transmission lines are modeled using a distributed parameters equivalent circuit when actual currents and voltages are required at varying points along the line. For stability studies normally only the current and voltage values at the buses are of interest. Therefore the nominal-π model is used for long lines also. The minor discrepancies in voltage and current resulting from its use are normally insignificant (17). For greater accuracy in long line models, a method called line sectionalization is used. Sectionalization

subdivides long lines into sections with common characteristics. The parameters for each section are then computed and used in the program (2).

Direct Current line model. According to Arrillaga (2), D.C. transmission lines are modeled by placing a resistance in series between two buses. The resistances of the line, the smoothing reactors, and the converters are all lumped together. Major concerns with D.C. transmission are minimizing the required reactive power and the line losses, which are related to the current. Therefore, voltages on D.C. lines are usually very high to keep currents low.

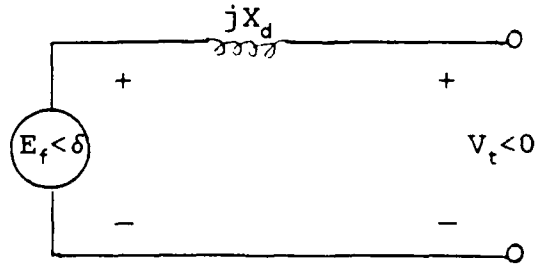
A.C. - D.C. conversion models. Arrillaga (2) states that whether or not the A.C. - D.C. converters are modeled in detail depends on the type of study being conducted. During steady state studies they can be modeled by incorporating their effects into the transmission line, as discussed above. During a system disturbance this simple model does not apply. Arrillaga (2) presents a detailed converter model and demonstrates that during abnormal operating conditions at least six differential equations are required for its representation. The important point is that each

converter modeled adds six differential equations to the solution matrix, increasing solution complexity and time.

Synchronous Machine Models

Figure 1 is the voltage behind reactance model for synchronous machines (17). It uses an internal excitation voltage ($E_f<\delta$) and the machine's synchronous reactance (X_d) to produce the steady state terminal voltage (V_t) and current.

During a stability study, the synchronous reactance is replaced with the transient or subtransient reactance. Using the new reactances, new current, voltage and power angle values are calculated for each incremental time step during the study.



$E_f<\delta$ = Excitation voltage

$V_t<0$ = Terminal voltage

X_d = Synchronous reactance

Figure 1. Basic synchronous machine model

During a fault, synchronous machines must be checked for two main problems. The first is whether or not the machine returns to synchronous operation, which is determined by the power angle. The second is whether or not the current in the windings will damage the machine. According to Arrillaga (2), the basic model is sufficiently accurate for transient stability studies of less than one second in duration, or where the system is not operating near its limit or critical clearing time. For steady state and dynamic stability studies a detailed model must be used to include the effects of automatic machine controllers. The detailed model can increase the number of equations in the system solution from three to nineteen (19).

The real (P) and imaginary (Q) power produced by a synchronous machine can be expressed as:

$$P = \frac{|E_f|\ |V_t|}{|X_d|} \sin\delta \qquad (1)$$

and

$$Q = \frac{|E_f|\ |V_t|}{|X_d|} \cos\delta - \frac{|V_t|^2}{|X_d|} \qquad (2)$$

The power can be controlled by changing $|E_f|$, $|V_t|$, $|X_d|$, or the power angle δ. During transient stability studies the synchronous reactance (X_d) changes and possibly the terminal voltage (V_t) depending on the fault location. The real power (P) and excitation voltage (E_f) initially remain constant. Therefore, the total change, resulting from the disturbance, must be accounted for in the power angle (δ) and the terminal voltage (V_t). If the power angle gets too large the machine will lose synchronism and drop off line, causing another system disturbance. This could trigger a chain reaction possibly causing the whole system to fail.

Following the first few cycles of a disturbance automatic voltage control systems sense the changes in voltage and frequency and begin to influence the generators. The size of the excitation voltage (E_f) is controlled by the excitation current in the rotor windings. Figure 2 is a composite block diagram modeling the excitation control system. This model can add up to eleven differential equations per generator to the system solution.

Large generators in a system are usually driven by water or steam, changing their rotation rate requires more time than changing the excitation voltage. The

rotation rate is controlled by governors and their effects do not appear until several cycles into a fault. Figure 3 is a generalized speed governor block diagram for a turbine generator. The generator's rotation is sensed and the flow of water or steam to the turbine is adjusted based on the error between the actual value and a reference value. This model adds up to five differential equations in to the system solution.

Transformer Model

A single phase transformer is modeled by a series impedance between its primary and secondary sides with the voltages related by the turns ratio. According to Arrillaga (2) this model cannot be extended to a three phase system. Three phase transformers have four basic configurations: wye-wye, delta-delta, wye-delta, and delta-wye. The proper model to be used depends on how the transformer is grounded. Arrillaga (2) develops matrix representations for the different configurations, but concludes that normally not enough detailed information is known to use the general models. Therefore, three phase transformers are modeled using known parameters and symmetrical components models. Figure 4 summarizes the symmetrical components models for three phase transformers.

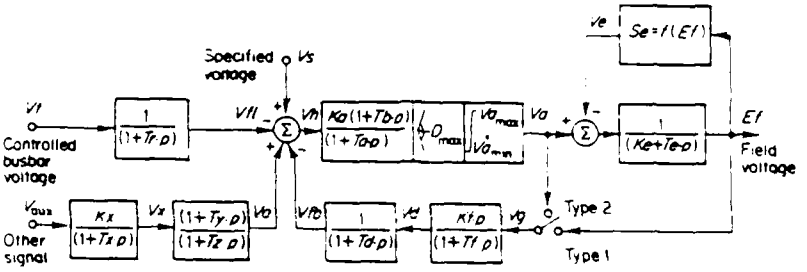


Figure 2. Composite excitation control block diagram (2)

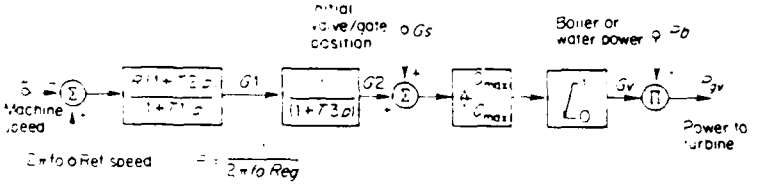


Figure 3. Composite speed governor block diagram (2)

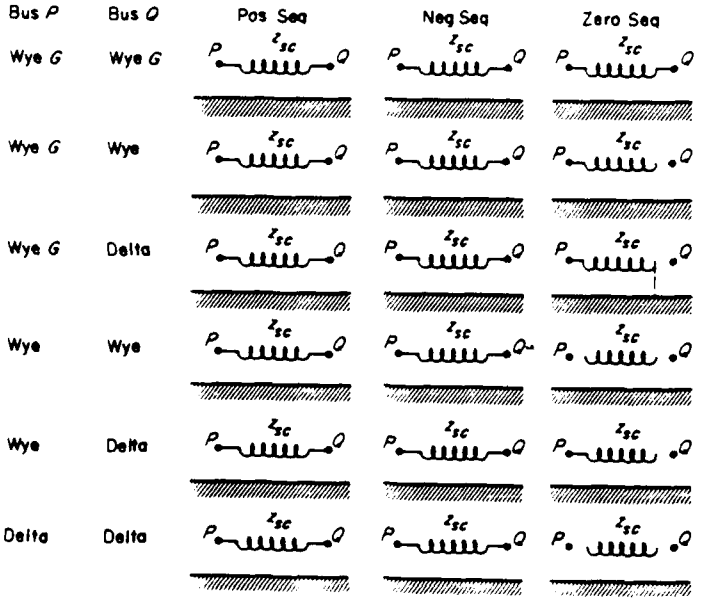


Figure 4. Symmetrical components networks for common three phase transformer configurations (2)

System Models

Loads. Static loads, loads without synchronous machines, remain relatively unchanged during faults and can be modeled as constant loads. Dynamic loads however may be very sensitive to system voltage changes. Arrillaga (2) presents the following differential equation to model the relationship between the power and the voltage for large dynamic loads.

$$\frac{dV_{(overall)}}{dt} = \sum_{j=1}^{n} \left(\frac{dV_j}{dt}\right)(P_j) \Big/ \sum_{j=1}^{n} (P_j) \qquad (3)$$

This equation must be incorporated into the system of simultaneous equations solved by the stability program.

Faults. Fault models depend on the type. For shunt faults, the bus where the fault is located is connected through a low impedance to ground. Series faults are simulated by replacing the normal line impedance with a very large impedance. Fault modeling requires the modification of the impedance or admittance matrix depending on the solution algorithm. Sparse matrix techniques minimize the additional solution time required to modify these matrices.

Line switching. Removing a line during a study can be handled like a fault by changing its impedance. The addition of a new line can be implemented in two ways (2). It can be added during the study, in which case the impedance or admittance matrix must be reordered and recalculated; or it can be added at the beginning of a study with a very high impedance. The second method requires only modification of the impedance or admittance matrix, and is the preferred method.

Machine Switching. Machines can be switched in and out of the system two ways. At the beginning of the program a dummy bus can be added between the machine and the network. To remove a machine a current can be injected at this bus nullifying the machine's effect. With this method the machine can be easily switched in and out of the system during the study. A second method is to add or remove the machine from the system during the study. This method requires network reconfiguration and recalculation of all matrices. Using the second method adding or removing to a large system is very costly in computational effort.

Machines against the system. The models presented so far only simulate the effects of individual changes at

the machines. They do not simulate the dynamic changes in the system. Equations (4) and (5), the swing equations, relate power at the machine, system base frequency (f_o), synchronous frequency (w), bus power angle(δ), and the machine's angular momentum (Mg) to time (2;8;17).

$$\frac{dw}{dt} = \frac{1}{Mg}(P_m - P_e - D_a(w - 2\pi f_o)) \qquad (4)$$

$$\frac{d\delta}{dt} = w - 2\pi f_o \qquad (5)$$

D_a = Damping coefficient

Mg = Angular momentum

P_m = Mechanical power

P_e = Electrical power

During a fault the bus voltage and power angle change. The machine attempts to respond by changing speed, but inertia does not allow an immediate change, causing a large surge in current. The current immediately after a fault is shown in Figure 5. The stability study must solve the swing equations at all machine buses to calculate the currents and related power angles at each bus. Once these parameters are known, system planners can provide protective devices for the

system to insure no equipment is damaged.

System Solution Techniques

Introduction

Computers are essential for a large network stability study. The requirements to recalculate impedance parameters, reconfigure the network, and solve large systems of simultaneous non-linear and differential equations make hand calculation impossible.

The difficulty with programming a stability study solution is solving the large number of simultaneous differential and algebraic equations associated with the network models in an efficient and accurate manner. Transient stability studies are more complex since they require three phase representation as opposed to steady state studies which can use one phase. Therefore, symmetrical components must be used to model the system, increasing solution complexity.

Symmetrical components divide currents and voltages into three independent components positive, negative, and zero. Each component has an associated network which must be solved, yielding a portion of the total voltage or current. The components are then combined to get the total current or voltage. Arrillaga (2) provides an

excellent derivation of symmetrical components models and equations for an A.C. - D.C. network. The derivation is too long for presentation here, but to give an example of how the number of equations increase when symmetrical components analysis is used, each A.C. - D.C converter requires a minimum of six differential equations for steady state analysis, but requires twenty-six equations for transient analysis.

To reduce the number of equations during large network studies, groups of machines can be combined and modeled as a single load (17). The distance between the fault and loads dictate whether this is acceptable.

Traditionally, load-flow and stability studies have been solved using either the Gauss-Seidel method or the Newton-Raphson method.

Gauss-Seidel Method

The Gauss-Seidel method solves each equation in the system in turn, by assuming a value for each variable, solving the equation, comparing the results with known quantities, and then adjusting the assumed value associated with that equation. Each equation is solved twice in this manner before proceeding to the next equation.

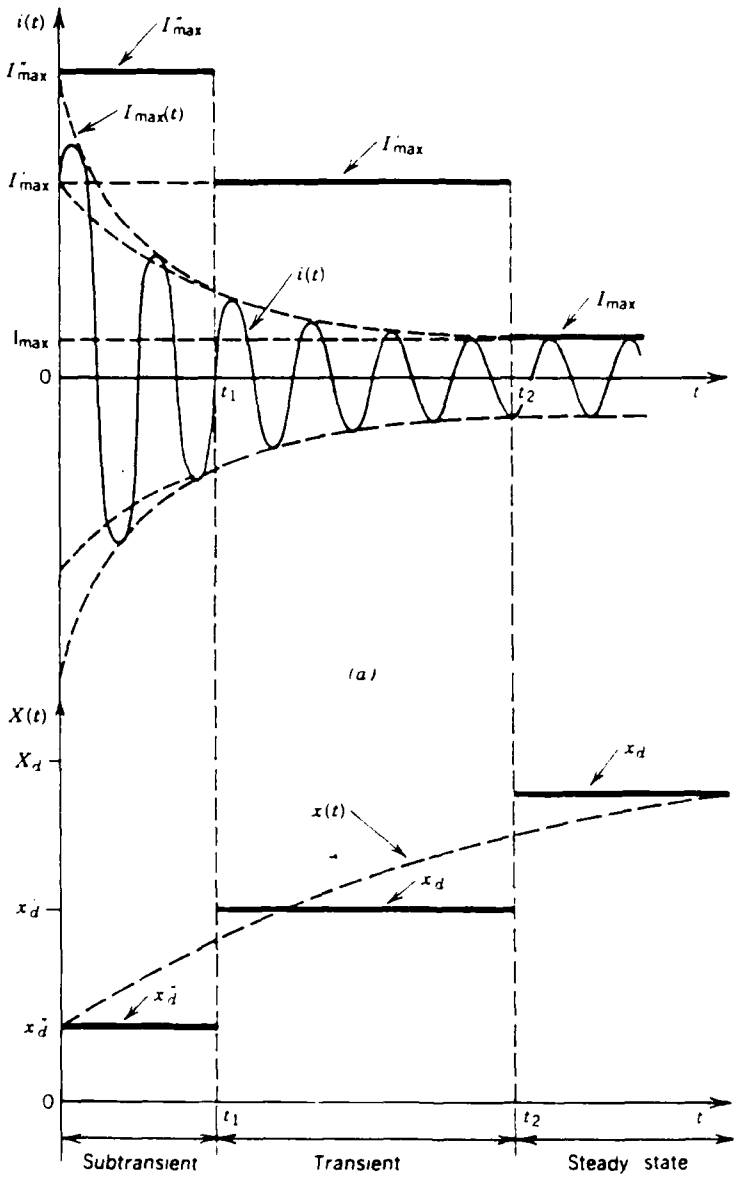


Figure 5. Balanced fault current and reactance for one phase of a synchronous machine versus time: (a) instantaneous fault current without D.C. offset, (b) reactance with stepped approximation (8).

Each equation adjusts one variable, the adjusted value is then used in the succeeding equations. This process is repeated until a sufficiently accurate solution is found or a predetermined number of iterations has been performed. If the iteration count is exceeded it is assumed the system is not converging, an adjustment to the starting values must be made, and the process repeated (2;7;8;17).

The Gauss-Seidel method converges very slowly compared to the Newton-Raphson method. It is also very sensitive to network configuration, series capacitance and negative resistances according to Gonen (8). For these reasons the Gauss-Seidel method has primarily been replaced by the Newton-Raphson method.

Newton-Raphson Method

Like the Gauss-Seidel, the Newton-Raphson method starts with an assumed solution. The assumed solution is the most critical step in the method according to Arrillaga (2). If the assumed solution is poor then the solution may diverge or converge at an incorrect solution. This problem is eliminated in stability studies by using the results from a previous load-flow study as the initial values. For load-flow studies other techniques are available, see reference (2).

Once initial conditions are established, the Newton-Raphson method uses a Taylor series expansion to approximate the next value (8). The Newton-Raphson method is used to solve Equation (6) for the A.C. network during stability studies (8).

$$\begin{bmatrix}\delta P\\ \delta Q\end{bmatrix} = \begin{bmatrix}J1 & J2\\ J3 & J4\end{bmatrix}\begin{bmatrix}\delta\\ \delta|V|\end{bmatrix} \qquad (6)$$

The off-diagonal and diagonal values for submatrix J1 can be evaluated as:

$$\frac{\partial P_i}{\partial|V_j|} = |V_i||Y_{ij}||V_j|\sin(\theta_{ij}+\delta_i-\delta_j) \qquad i\neq j \qquad (7)$$

and

$$\frac{\partial P_i}{\partial|V_i|} = |V_i|^2|Y_{ij}|\sin\theta_{ij} - Q_i \qquad i=j \qquad (8)$$

The off-diagonal and diagonal values for submatrix J2 can be evaluated as:

$$\frac{\partial P_i}{\partial|V_j|} = |V_i||Y_{ij}|\cos(\theta_{ij}+\delta_i-\delta_j) \qquad i\neq j \qquad (9)$$

and

$$\frac{\partial P_i}{\partial |V_i|} = -\frac{P_i}{|V_i|} + |V_i||Y_{ij}|\cos\theta_{ij} \qquad i=j \qquad (10)$$

The off-diagonal and diagonal values for submatrix J3 can be evaluated as:

$$\frac{\partial Q_i}{\partial \delta_j} = -|V_i||Y_{ij}||V_j|\cos(\theta_{ij}+\delta_i-\delta_j) \qquad i\neq j \qquad (11)$$

and

$$\frac{\partial Q_i}{\partial \delta_i} = -|V_i|^2|Y_{ij}|\cos\theta_{ij} + P_i \qquad i=j \qquad (12)$$

The off-diagonal and diagonal values for submatrix J4 can be evaluated as:

$$\frac{\partial Q_i}{\partial |V_j|} = |V_i||Y_{ij}|\sin(\theta_{ij}+\delta_i-\delta_j) \qquad i\neq j \qquad (13)$$

and

$$\frac{\partial Q_i}{\partial |V_i|} = |V_i||Y_{ij}|\sin\theta_{ij} + \frac{Q_i}{|V_i|} \qquad i=j \qquad (14)$$

The Newton-Raphson method is the most popular since it converges quickly, usually in two to five iterations; is very reliable, given a good initial approximation; is insensitive to system configuration, series capacitance and negative resistances; has a convergence rate relatively independent of network size; uses either rectangular or polar coordinates; and uses the admittance matrix which is usually very sparse, requiring less storage (2;8). Sparse matrix techniques can also be applied easily to the method, significantly reducing the number of operations required for solution and therefore reducing overall computation time and cost (2).

Figure 6 presents a general flow diagram for a transient stability study program involving both A.C. and D.C transmission line. L_l represents the network admittance matrix and can be either time-variant or time-invariant. L_g represents the connection matrix for the D.C. converter valves and can also vary with time (2).

Examination of the flow diagram in Figure 6 gives an indication of the complexity of the problem. The block labeled "Solve network equations for one step" represents the heart of in the program. It is there that the Newton-Raphson method is employed to simultaneously solve the algebraic and differential equations.

Summary

Modeling a power system network, to include all components of the system, results in a large number of simultaneous non-linear algebraic and differential equations. The complexity of the solution is further increased by the introduction of symmetrical components required to analyze unbalanced three phase faults. The complexity of the detailed models and the size of a large system prohibit all machines from being individually modeled in the solution, otherwise the time and computational costs would be to great. Therefore parts of the network are usually modeled by combining the generation and loads at a bus and representing that bus as a constant load. The Newton-Raphson method for solving large systems of simultaneous equations combined with sparse matrix techniques make it possible, using computers, to perform fault analysis on the reduced system. Using a vector supercomputer to perform the repetitious operations at a much faster rate than standard computers would allow the system to modeled in greater detail. The result would be more accurate system values and better system utilization.

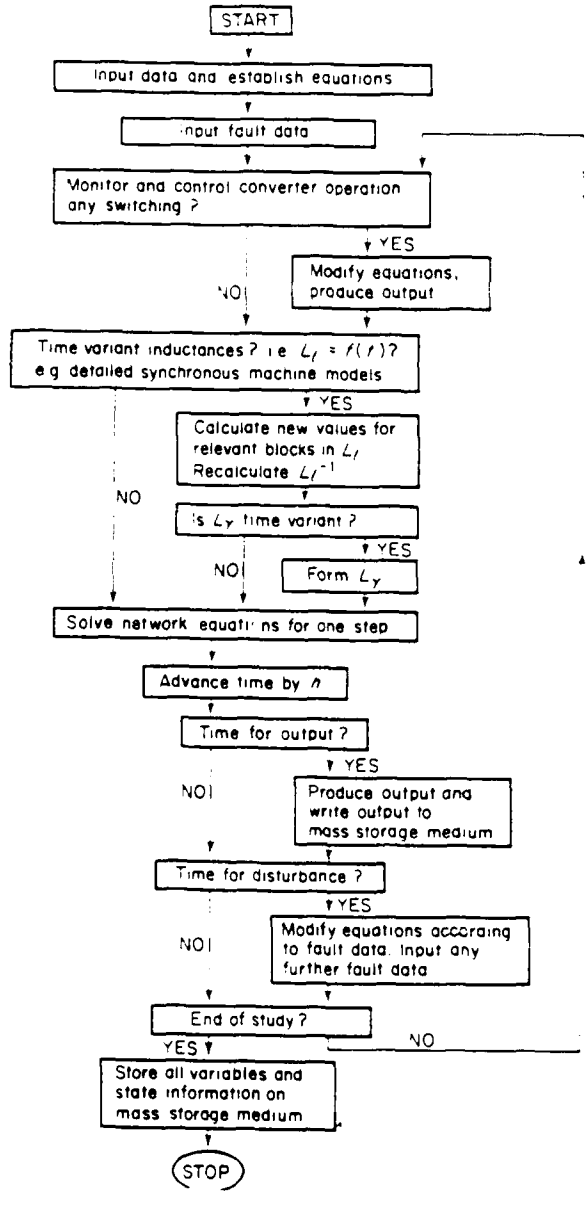


Figure 6. General flow diagram for the solution of an A.C. - D.C. transient stability program (2)

CHAPTER III

WSCC POWER STABILITY PROGRAM

Introduction

The power stability program optimized for this study was developed and is used by the Western Systems Coordinating Council (WSCC) and its members. The program solves model equations similar to those introduced in Chapter II. A stability study can start either from a steady state condition or from a previous stability study, permitting users to examine the effects of different actions without rerunning the complete study. The program simulates switching for transmission lines, loads, and generation and simulates different types of faults. After a study, output data is available in several formats for detailed analysis later (19).

Program Organization

Conceptually, the program can be divided into four broad phases. The first phase initializes the electrical

state of the network. The second phase inputs any system modifications before starting the time related solution. The third phase simulates the fault or disturbance for a given time period. It calculates the bus voltages and angles for each time step and saves them as intermediate results for output later. The fourth phase allows further system modification, restart of the study with different inputs or termination of the study.

Network Initialization

The program loads the initial network electrical state from a history file. The history file is produced either by an earlier load-flow study or a previous stability study. It provides electrical data for all buses and components. After reading the history file, the buses with load netting are identified and their values entered. Load netting combines the load and generation at a bus to form a constant load reducing the number of equations. Next, dynamic load models are identified and their variables initialized. Following this, the generator, exciter, governor, and static var device characteristics are entered, identifying their model types and initializing their variables. Table 1 summaries the number of models available in the program for different devices. Once these characteristics are

entered, the data describing the relays in the system and defining their limits of operation is inputted. Finally, the variables which control automatic load shedding are read in. After reading the history file the program is ready to accept the initial system modifications and start the time-related solution.

Initial System Modification

After program initialization, any system changes to be implemented before the fault, and the fault or disturbance to be studied are entered. Table 2 summarizes the user directed system modifications and the disturbances which can be simulated. User modifications simulate actions directed by a distribution control center for routine system maintenance or in response to a fault. Next, the start time, end time, and time step to be used during the study are entered. The program is now ready to solve the system equations for the new bus conditions resulting from the modifications and disturbance.

Fault or Disturbance Simulation

The program's third phase applies the disturbance to the system and simulates the system response.

Table 1. Models in the WSCC Power Stability Program (19)

Synchronous Generators
- a) Constant voltage behind reactance - one model
- b) Detailed model - one model

Generator Excitation Systems - eleven models

Power System Stabilizers
- a) Three models
- b) Four sources of model input
  1) Terminal voltage
  2) Generator slip
  3) Generator accelerating power
  4) Bus frequency deviation

Governor Systems - four models

Turbines - seven models

Induction Motors - one model

Load Representation and Netting
- a) As constant power or current or a combination
- b) As a function of voltage and frequency by bus or by area

Load Shedding and Restoration - two models

Static Var Devices - six models

D.C. Lines and Controls - sixteen features can be modeled

Relays - five models

Remote Relay Operation - five models

Table 2. Disturbances and User Directed Modifications in the WSCC Power Stability Program (19)

Disturbances
a) Three Phase Fault
b) Line to Ground Fault
c) Line to Line Fault
d) Generator Tripping
User Modifications
a) Disconnect Buses and Lines
b) Reconnect Buses and Lines
c) Modify Shunt Impedance
d) Modify Line Impedance
e) Flash and reinsert Capacitor Gaps

Additional system changes, such as power distribution control center directives, can be entered during this phase. An alternating solution algorithm is used to calculate new D.C. and A.C. system bus values. The algorithm alternatively solves the A.C. and D.C. systems, while holding the other system values constant. It is implemented in the following manner. First the status of the relays and lines are checked. If a relay's status was changed or a line connected/disconnected, a subroutine to modify the system matrix tables is called. After modifying the system matrices, new bus values for the A.C network are found, assuming constant D.C. values. Then holding the new A.C. values constant, the D.C. system is solved for new bus values. After calculating new A.C. and D.C. bus values, routines to check for relay

action or load shedding as a result of the modifications or the disturbance are called. If relays operated or loads were shed then the process is repeated beginning with matrix modification. This process is continued until the system is stable, then the bus values are saved for use at the beginning of the next step.

After the system is stable, the fault or disturbance is applied and the time related solution is started. The main program calls a subroutine (TSCALC) to control the iterative solution algorithm.

TSCALC first calls a set of routines (DCAUXY, DCMOD, DCTRAN, and DCSIMP) to first update the D.C. the model variables and then solves the D.C. model and system equations. After solving the D.C. system for the first time step, the program solves the A.C. system.

The A.C. solution first checks whether any generators were disconnected as a result of system modifications. If a generator's status changed the generator matrices are modified to reflect the change. Then, subroutine GOVERN is called to update the governor variables, and TSCALC evaluates the governor differential equations. Next, subroutine EXCITE updates the excitation system variables, and TSCALC evaluates the generator differential equations. Similarly, the variables and equations for the induction motors and

loads are updated and solved.

After updating the A.C. models, the subroutine SOL is called to solve the A.C. system's simultaneous equations for new bus voltages and power angles. TSCALC then calculates new bus frequencies. These frequencies are used as model inputs during the next time step. The new bus values are then compared to specified or known bus values. If the new values are within a given tolerance of the specified values the program continues with the next check. If the values are not within the tolerance the time step is reduced and the D.C. and A.C. systems are resolved using the smaller time step. If after a designated number of solution attempts, the values still do not converge, the study is aborted and an error message is provided.

The next step in the algorithm checks for automatic or user controlled network changes. If the network is changed by relay action, load shedding, line switching, or generator switching the system tables are rebuilt and the effects at each bus calculated. If no network change takes place, the solution time is compared to the end time. If they are equal the program continues with the next phase. Otherwise, the solution time is advanced and the D.C. subroutines are called again to begin solving the system for the next time step.

Study Restart or Termination

Once the first end time is reached, the study can continue with new conditions and times or terminate and output the results. If the study continues new system modifications and times are entered and the solution process repeated. If the study terminates output is available in several formats. Table 3 summarizes the different output formats available.

Table 3. Program Output Formats (19)

Rotor angle and Electrical Power as a Function of Time
Bus Voltage Report
Generator Report
Relay Operation Report
Line Flow Report

Equation Solution

The non-linear algebraic equations are solved in the subroutine SOL using the Newton-Raphson method and sparse matrix techniques. An equation similar to Equation (6), page 26, is solved for the power angle (δ_s) and voltage

($\delta|V|$) changes at each bus. Instead of inverting the Jacobian matrix, it is factored into upper right and lower left matrices. Then, forward and backward substitution is used to find the difference values. The updated Jacobian matrix values for each iteration are found directly by solving equations similar to Equations (7) through (14), pages 26 to 27.

Laplace transforms were used to integrate the differential equations associated with the models during program development. The inputs are assumed to be ramp functions unless switching occurred. At switching times the inputs are assumed to be step functions. The inverse Laplace transform expressions were programmed and are solved either in SOL or TSCALC (19).

Each model in Table 1, page 34, is identified by a flag variable, the program uses the flags to jump to the appropriate model equations in the code. Large models, like the excitation model in Figure 2, page 18, are solved one block at a time. The first block's inputs are the latest system bus values, its outputs are calculated and used as inputs to the next block. This process continues through the diagram until the complete model is evaluated.

Summary

Due to the large number of models and associated equations, the power stability program is very large and complex, it contains approximately fifty thousand lines of code. This chapter presented a broad outline of the overall program focusing on the subroutine TSCALC, which solves the model and system equations. It is shown in Chapter VII that the majority of the solution time is spent in SOL solving the A.C. network. This is to be expected since very few D.C. transmission lines are currently in use.

The repetitiveness of the solution, the large sparse matrices associated with power networks, and the large number of floating-point operations in the program to make it a good candidate for implementation on a vector supercomputer.

CHAPTER IV

SUPERCOMPUTER CLASSIFICATION

Introduction

Kai Hwang and Faye A. Briggs (9) in their book Computer Architecture and Parallel Processing, state that supercomputers are designed for high computational speed, possess large, fast main and secondary memories and make extensive use of parallel structured hardware and software. Supercomputers are essential for scientific and research computing which require millions of floating point operations. Their computational speed can greatly reduce the time and cost associated with modern simulations, allowing researchers to use more detailed and precise models. The first section of this chapter discusses classifications of supercomputers according to instruction and data flow. The second section of the chapter discusses computer pipelining in detail since the Cray Y-MP is a highly pipelined multiprocessor computer system.

Classification by Instruction and Data Flow

Computers have traditionally been classified by how instructions and data move through the system. The classification system separates computers into four categories: Single Instruction and Single Data, Single Instruction and Multiple Data, Multiple Instruction and Single Data, and Multiple Instruction and Multiple Data.

Single Instruction and Single Data (SISD)

Single Instruction and Single Data computers execute one instruction on a single data stream. Also known as uniprocessors, they consist of a control unit, a processing unit, and a memory unit. As the name implies the control unit fetches one instruction, decodes the instruction, fetches the required operands, and then instructs the processing unit to perform the operation. Uniprocessors may contain more than one functional unit and may be pipelined for increased speed. Pipelining will be described in more detail later in this chapter.

Single Instruction and Multiple Data (SIMD)

Single Instruction and Multiple Data computers execute one instruction on several data streams. Also

known as array processors, they consist of one control unit, two or more processing units, and a shared memory unit. According to Hwang and Briggs (9), spatial parallelism is achieved when the control unit instructs all processing units to perform a single operation on their own data stream.

Multiple Instruction and Single Data (MISD)

Currently, no Multiple Instruction and Single Data computers exist. Were a system to be built, it would have two or more processors, with unique instruction sets, operating on a single data stream. According to Hwang and Briggs (9), many people consider this category impractical and no work is currently being done in this area.

Multiple Instruction and Multiple Data (MIMD)

Multiple Instruction and Multiple Data computers execute multiple instruction streams on multiple data streams. They are also known as multiprocessors and multiple computer systems. Hwang and Briggs (9) state that these systems achieve asynchronous parallelism by having two or more approximately equal interactive processors working independently. Each processor has its

own instruction stream and data stream but share resources such as main and secondary memories, databases, and input/output channels. The processors may work on separate problems or cooperate on a common problem. The Cray Y-MP is a pipelined multiprocessor system and will be discussed in detail in Chapter V.

Pipeline Computers

Computer pipelining exploits temporal parallelism by overlapping computations, central processor unit operations, and different functional unit operations. The different functional units share expensive system resources which are controlled by a central processing unit.

Pipelining

A pipeline is the interconnection of two or more specialized functional units. Each unit performs a specific independent subtask of a larger task or operation each pipeline cycle. The configuration of the pipeline can be controlled either through hardware or software. A pipeline cycle is the time required for the slowest pipeline unit to complete its task.

Hwang and Briggs (9) give a good example of an

instruction pipeline using the instruction fetch and execute operation of the central processing unit. The instruction fetch and execute function can be divided into four pipeline tasks, for example: instruction fetch (IF), instruction decode (ID), operand fetch (OF), and execution (EX). The instruction pipeline consists of four units, one for each task. The IF unit gets a new instruction and outputs it to the ID unit during one pipeline cycle. During the next cycle the IF unit gets a second instruction while the ID unit decodes the first instruction. During the third cycle the IF unit gets a third instruction, the ID unit decodes the second instruction and the OF unit fetches the required operands for the first instruction. During the fourth cycle the IF, ID and OF units perform their functions on the applicable instructions while the EX unit executes the first instruction. After four cycles, the pipeline is full and a new instruction can then be executed every pipeline cycle as opposed to every instruction cycle which equals four pipeline cycles.

With pipelines the functions of the central processing unit, the input and output devices, and the memory management devices can be overlapped to increase system performance. Pipelines can be optimized in hardware for specific types of operations such as scalar

or vector operations further enhancing system performance.

Pipeline Computer Classification

Hwang and Briggs (9) present two classification schemes for pipeline computers. The first classification scheme is based on three processing levels: arithmetic, instruction, and processor.

Arithmetic pipelining. Arithmetic pipelining divides the arithmetic logic units of a computer into pipelines optimized for different data formats. An example of an arithmetic pipeline is the addition of two floating-point numbers with the same sign. The pipeline can be configured with four stages: a base alignment stage, a coefficient add stage, a coefficient sum normalization stage, and a result round-off and sign attach stage. During the first pipeline cycle the base alignment stage, stage 1, operates on the first set of operands. During the second cycle stage 1 operates on the second set of operands while the coefficient add stage, stage 2, operates on the first set of operands. During cycle three, stages 1 and 2 operate on the third and second set of operands respectively, while the coefficient sum normalization stage, stage 3, operates on

the first set of operands. During the fourth cycle the first three stages each operate on their particular set of operands while the result round-off and sign attach stage, stage 4, produces the final result for the first set of operands. During the fifth and subsequent cycles stage 4 outputs a new result each cycle until all operands have been processed.

Instruction and processor pipelining. Instruction pipelining, overlaps fetching, decoding, and executing of instructions as described earlier. Processor pipelining interconnects multiple processors. Each processor, in turn, gets data from a common memory, operates on it and then returns the data to memory.

The second classification scheme is based on pipeline configuration and control strategies.

Unifunctional and Multifunctional. Unifunctional pipelines are designed to perform a single type of function. Multifunctional pipelines are designed to perform several different functions either simultaneously or at different times. Multifunctional pipelines can be reconfigured depending on the current requirements and are further sub-classified as static or dynamic.

Static and Dynamic. A static multifunctional pipeline is under software control and is incapable of frequent functional reconfiguration. It is most efficient for very repetitive operations. A dynamic multifunctional pipeline is under hardware or firmware control and is capable of frequent functional reconfiguration. It requires a very elaborate control network and is more complicated and expensive than a static pipeline.

Scalar and Vector. Pipelines are also classified as scalar or vector pipelines. A scalar pipeline is designed to execute sequential DO loops efficiently by prefetching instructions and data. A vector pipeline is designed to perform vector operations efficiently under hardware or firmware control.

Pipeline Computer Architectural Classifications

Hwang and Briggs (9) present two architectural configurations for pipeline supercomputers: vector supercomputers and attached array processors.

Vector supercomputers. Vector supercomputers are designed to perform operations on data sets which can be treated as vectors, such as matrices. They are best

suited for programs requiring repeated identical operations on a large number of operands. This type of repetitious processing exploits temporal parallelism. Full matrices give the best performance improvement, but developments in sparse matrix programming have greatly improved the performance of sparse matrix operations.

Vector pipelines require additional overhead, known as setup time and flushing time, to allow vectorization. Setup time is the time required to route the operands among functional units in the pipeline. Flushing time is the time between decoding an instruction and the appearance of the first result. The difference between sequential computer and vector computer flushing times is usually small. The run time difference between sequential and vector processing will more than offset the vectorization overhead for programs which vectorize.

The vector length when vector processing time is less than scalar processing time for the same operation is known as the vector break-even length. It is very machine dependent and critical when dealing with sparse vectors. If the vector break-even length is longer than the sparse vector length then scalar processing will result in a faster run-time.

Attached array processors. Attached array

processors are designed to improve the performance of a host computer by providing vector-processing capabilities and improved floating-point operations. Attached array processors are controlled by the host computer, but operate independently after receiving the required data; this allow the host computer to perform other operations until the attached array processor is finished. Attached array processor hardware is cheaper than vector computer hardware, but the software is usually application specific and can be expensive.

Pipeline Computer Attributes

Pipeline computers are especially well suited for many types of scientific and research related activities. They were specifically designed to perform fast repetitive calculations through pipelining and vectorization. Current systems employ multiple scalar and vector pipelines to increase speed and flexibility. The ability to cascade pipelines, called chaining, allows results from one pipeline to be used immediately as operands to another pipeline. This reduces the number of memory accesses required. Chaining is only limited by the number of vector registers in the system.

Pipeline Computer Performance Degradation

Pipeline computer performance can be lowered by several factors. Resource conflicts may arise when two or more pipeline stages attempt to access the same shared resource simultaneously, resulting in one process being delayed. Data dependencies in the code may prevent the code from becoming vectorized and force it to be implemented sequentially. Recursion, storing the results of an operation immediately back into one of the operand registers, can also cause considerable program degradation.

Summary

According to Hwang and Briggs(9), pipelined uniprocessor systems currently dominate the supercomputer market in both the scientific and business areas. They have lower costs and better developed operating systems allowing them to achieve better resource utilization and higher performance. Array processors have been custom designed for specific applications but have a much lower performance to cost ratio. Additionally, the software required to utilize array processors is usually more complicated and expensive to produce. Multiprocessor systems are more flexible for general-purpose

applications without the programming difficulties associated with array processors. Current emphasis in computer development is on pipelined multiprocessors.

CHAPTER V

CRAY Y-MP8/864 SUPERCOMPUTER

The Cray Y-MP Computer Architecture

The Cray Y-MP8/864 is a Multiple Instruction and Multiple Data architecture computer system. Moore (14) states that the Y-MP at the San Diego Supercomputer Center has eight processor units, sixty-four mega-words of main memory, sixty-five giga-bytes of disk storage, and one hundred and twenty-eight million words of solid state storage. The computer's exceptional power and balanced memory architecture provides excellent computational power for large and complex problems.

The system is designed to support both scalar and vector programs and performs well on both parallel and non-parallel programs. It also operates in both multiprogramming and multitasking modes (14). The multiple processors allow the system to run one or more programs simultaneously, multiprocessing; or to simultaneously run parts of a single program, multitasking. Multitasking on the Cray Y-MP can be

implemented in three ways: macrotasking, parallel execution of large time consuming subroutines; microtasking, parallel execution of small sections of code; or autotasking, which allows the compiler to automatically perform both macrotasking and microtasking. Autotasking exploits the most levels of parallelism and is more efficient than microtasking (6).

Shared Functional Areas

Figure 7 is a system block diagram of the Cray Y-MP. One central processor is expanded to show the processor details. The eight central processor units in the system share four major functional areas: the real-time clock, the interprocessor communication section, the external input/output system, and main memory.

Real-time clock. The real-time clock is implemented using a sixty-four bit sequential counter. The counter advances once for each clock period and provides a common time reference for the system and a means to time program execution (6).

Interprocessor communication section. The processors communicate through the interprocessor communication section, it shares three sets of registers.

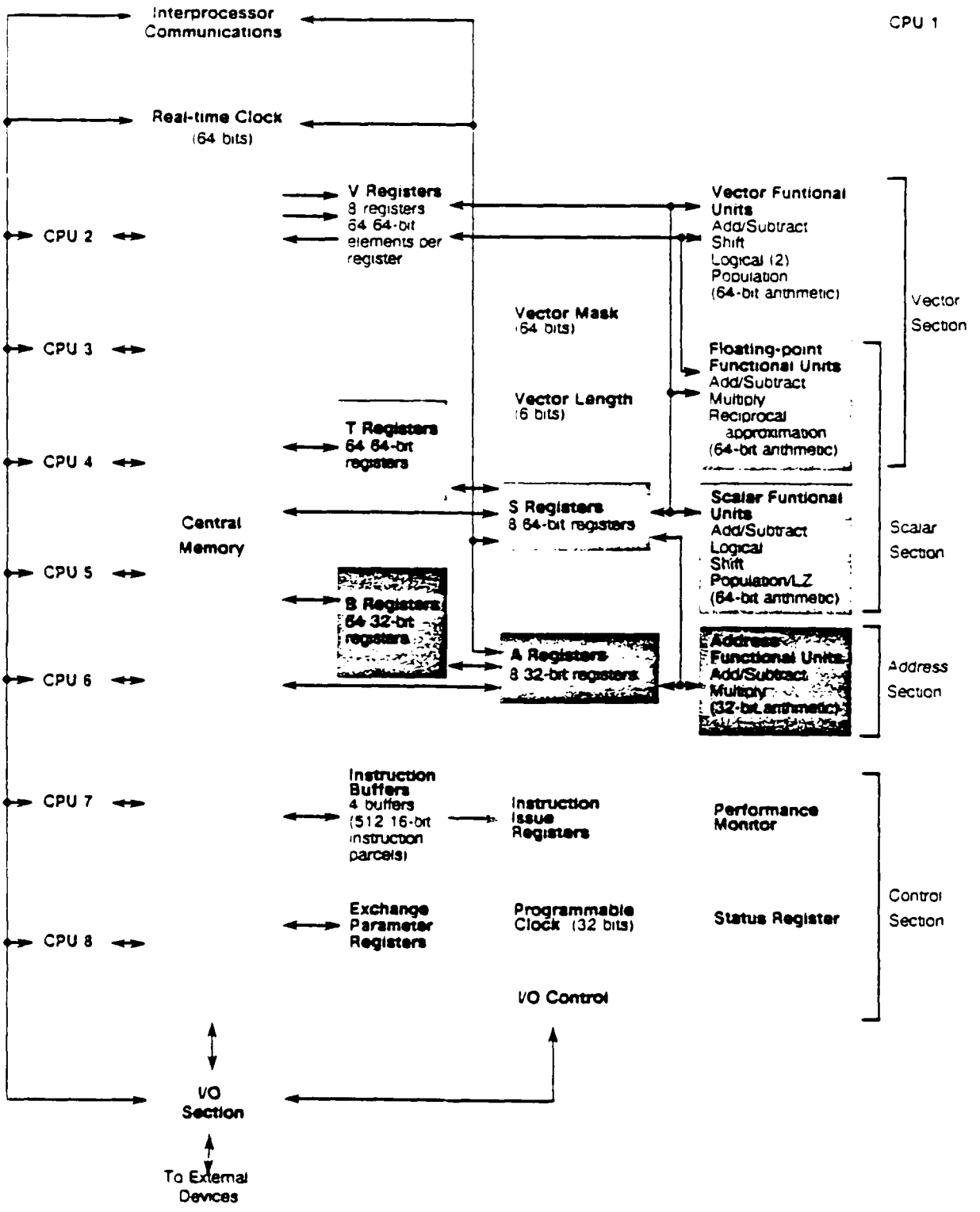


Figure 7. Cray Y-MP System Block Diagram (6)

The shared address registers (SB) transfer addressing information between processors. The shared scalar registers (ST) pass scalar data between processors. The shared semaphore registers (SM) contain the current status of each processor. These registers are critical for both multitasking and multiprocessing modes. During multitasking the processors use them to exchange data and timing information. During multiprocessing the operating system uses them to monitor the status of each processor (6).

External input/output system. The external input/output system performs data transfer from external devices attached to the system and the system's main memory. Three data transfer speeds are supported. For data transfer between main memory and external devices, there are eight channels rated at six mega-bytes per second and eight channels rated at one hundred mega-bytes per second. For data transfer between main memory and the solid state storage device there are two channels each rated at one thousand mega-bytes per second (6).

Main memory. The shared main memory contains sixty-four mega-words of interleaved memory, allowing almost concurrent access. The memory is divided into four

sections, each with eight subsections. Each subsection is divided into eight banks for a total of two hundred and fifty-six banks.

Each processor has four parallel input/output ports to main memory, one to each section. The ports can perform two operand fetches and one result store simultaneously. The ports also operate bidirectionally, allowing block reads and writes. Built into the hardware are conflict resolution and single-error correction and double-error detection circuitry. The data flows from memory into special registers in the processors, this allows the processor to prefetch data and helps increase the processing rate (6).

Central Processor Unit Characteristics

The Cray Y-MP system can be configured with form one to eight central processor units. The processors can operate either separately or together. Figure 7 identifies four major sections in each processor: a control section, an address section, a scalar section, and a vector section.

Control section. Each processor's control section is responsible for fetching, decoding and issuing the program instructions for that processor. The control

section contains four instruction buffers which can hold up to five hundred and twelve prefetched sixteen-bit instruction parcels. The program address register (P) contains the address of the next instruction to be executed. The buffers pass the instruction parcels to the instruction issue registers (NIP, CIP, LIP, LIP1) which decode and issue the instructions. The instruction issue registers are pipelined and can theoretically issue one instruction per clock period. The thirty-two bit programmable processor clock provides interrupts to the processor, for example a context-switch interrupt. The exchange parameter registers are specially designed registers to provide quick context switching. The status register contains the current processor status and is monitored by the computer operating system. The performance monitor tracks the number of instructions issued, the number of hold-issue conflicts, and the number of instruction fetch and memory conflicts during program execution. This information can be used to identify bottlenecks in program execution and to improve program performance (6).

Address section. There are eight thirty-two bit address registers (A) and sixty-four thirty-two bit intermediate address registers (B). The A registers

provide operands to the address functional units which calculate the address and index information. The A registers can also provide operands to the scalar registers. The B registers normally contain the addresses of data to be repeatedly accessed over a long period of time, eliminating the requirement to continually recalculate them. The address functional units are thirty-two bit, integer, pipelined arithmetic units which perform the addition, subtraction, and multiplication operations associated with addressing (6).

Scalar section. The scalar section has eight scalar registers (S), sixty-four intermediate scalar registers (T), and several scalar functional units. Each register is sixty-four bits long. All scalar program computations are done in the S registers and the scalar functional units. The S registers can provide operands to the scalar, logical, or floating-point functional units. They also can provide one operand to the vector functional units, or return information to the address registers. The T registers are buffers between the S registers and memory. They are used to prefetch data from memory for fast register to register transfer to the S registers during program execution. Data can be transferred between them and memory in blocks, reducing

data acquisition time. The intermediate address and scalar registers cannot directly access functional units. They are fast access buffers used to match the bandwidths of the memory and the processors, speeding up program execution.

The scalar functional units perform thirty-two bit addition, subtraction, logical, shift, and population/parity/leading zero functions. The floating-point functional units are shared by the scalar and vector sections. They perform sixty-four bit addition, subtraction, multiplication, and reciprocal approximation. The reciprocal approximation coupled with multiplication replaces the division operation (6).

Vector section. There are eight vector registers in each processor. Each vector register contains sixty-four elements and each element is sixty-four bits long. There are no intermediate registers in the vector section. The vector registers can directly access main memory, the vector functional units, and the floating-point functional units. In addition to the eight vector registers, there is a vector length (VL) register and a vector mask (VM) register. These registers are used during vector operations to store the vector length, and to perform vector masking operations. The vector

functional units perform sixty-four bit addition, subtraction, shift, logical (2 types), and population/parity functions (6).

Special Features of the Cray Y-MP

There are several special features that contribute to the Cray Y-MP's speed and parallel processing. The six nanosecond clock period makes it one of the fastest systems currently in production (18). The other features, discussed in more detail below, are its fully segmented pipelines, independent functional units, vectorization, vector chaining, and vector gather/scatter commands (6).

Segmentation and pipelines. The address, scalar, and vector functional units discussed earlier are all fully segmented. An operation is fully segmented when it is divided into segments; each of which, is completed in one clock period. Fully segmented pipelines allow a new operation to begin before the completion of a previous operation. Figure 8 demonstrates the operation of a five segment pipeline. Two operands enter the first segment of the five stage functional unit each clock period. After five clock periods a new result exits the unit each

clock period until all elements are processed. Segmented pipelines introduce parallelism into the processor itself and theoretically can produce a result every clock period after the pipeline is full.

In addition to the pipelines, several other operations in the Cray Y-MP are fully segmented. They are the exchange sequences used to perform a context switch, the memory reference sequences, and the instruction fetch and issue sequences. Overlapping these functions greatly reduces the time required to run a program by increasing the number of operations done concurrently by the processor (6).

Independent functional units. The functional units in the address, scalar, and vector sections are all fully segmented and completely independent. Their independence allows numerous operations to be simultaneously executed. At the completion of an operation the unit can be reconfigured and utilized again with minimal delay (6).

Vectorization. The Cray Y-MP's ability to vectorize the execution of an operation greatly reduces the overhead associated with loop execution. Vectorization sequentially performs an operation on a set of operands, a vector, through the execution of only one instruction.

If the operand set were to be processed in a loop, the processor would repeatedly issue the same instructions, greatly increasing the demands on the processor and reducing its ability to perform other operations. Also, a vector operation is a register to register operation, this reduces the number of memory conflicts and their associated delays. Vectorization also exploits the independent functional units' segmentation and the Y-MP's ability to chain vector registers (6).

Vector chaining. Before vector chaining was developed, a result had to be stored in memory before it could be used as an operand in another computation. Vector chaining allows the results from one vector operation to be immediately used as an operand for another operation. Thus a second operation can begin immediately after the first result is available from a previous operation. This eliminates the delay normally encountered waiting for completion of the first operation and storage of the results. Figure 9 illustrates an example of vector chaining three operations. As soon as the first element arrives from memory it is stored in V0 and also passed to the add functional unit, the results from the add operation simultaneously go to V2 and the shift functional unit, finally the results from the shift

operation concurrently go to V3 and the logical functional unit. If vector chaining were not allowed the add operation could not begin until V0 was completely filled and the results stored in memory. Similar delays would be associated with the shift and logical operations, greatly increasing the time required to complete the total operation. Chaining provides continuous data flow, reduces input/output time, and reduces the number of memory conflict delays (6).

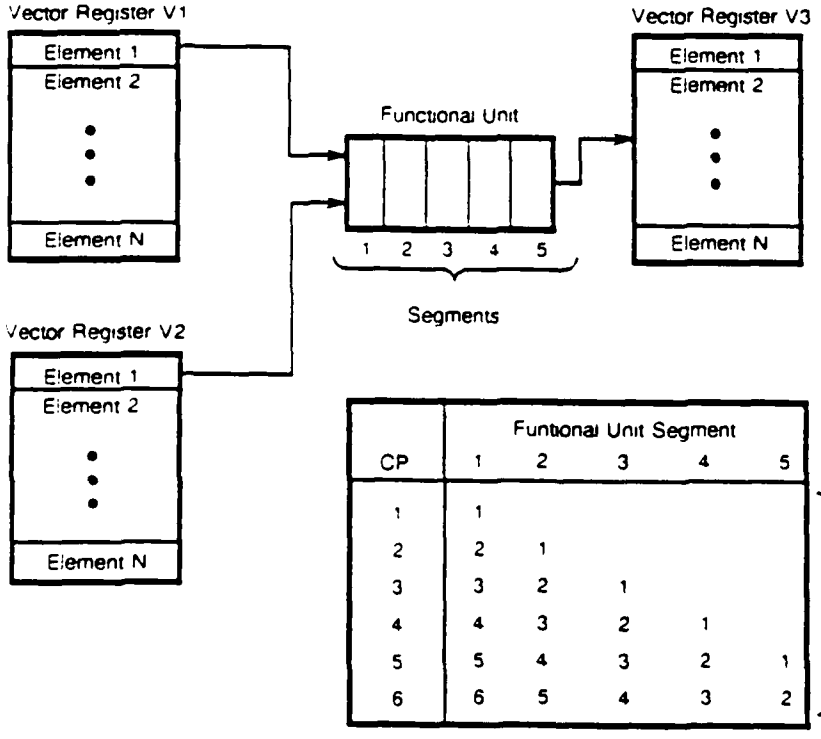


Figure 8. Functional Unit Segmentation (6)

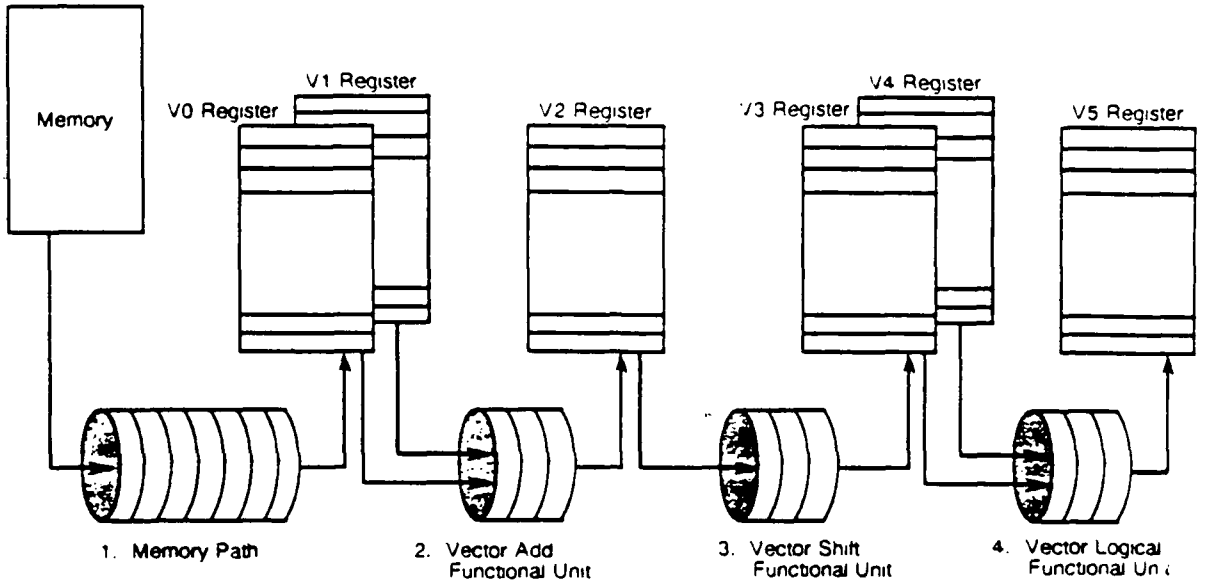


Figure 9. Vector Chaining of Three Operations (6)

Vector gather/scatter commands. The Cray Y-MP utilizes the vector gather/scatter commands to vectorize operations that use indirect addressing. This allows vectorization of operations whose operands have non-constant strides through memory. The commands use two vector registers. One register holds the index information while the other register holds the operand or result depending on whether it is a gather or scatter command. Figure 10 demonstrates how the gather command works. The operands are fetched form memory in the order specified in V0, the index register, and sent to register V1, the operand register. The scatter command is

demonstrated in Figure 11, it stores the results in V1 according to the index order in V0 (6). Since indirect addressing is frequently used in sparse matrix operations, these commands allow the compiler to vectorize code containing sparse matrix operations.

Table 4 summarizes the architectural characteristics of the Cray Y-MP. The next sections briefly describe some of the software available on the Cray Y-MP. The libraries described below provide optimized and vectorized routines that can be incorporated into programs to increase their execution speed. The timing utilities help identify lines of code for modification so that the system's high computational speed can be more fully utilized.

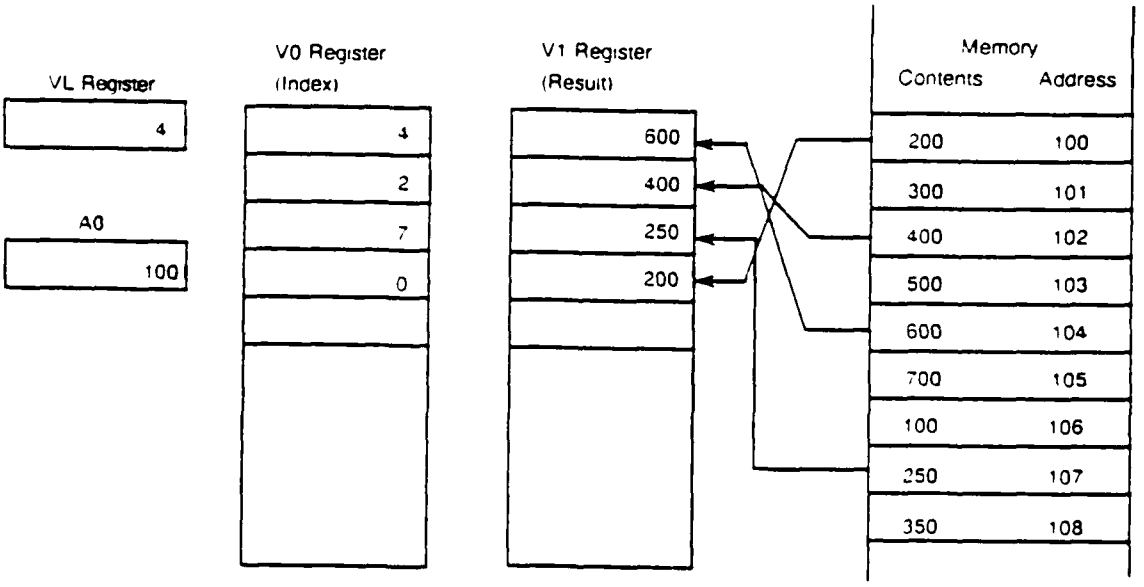


Figure 10. Cray Y-MP Gather Command (6)

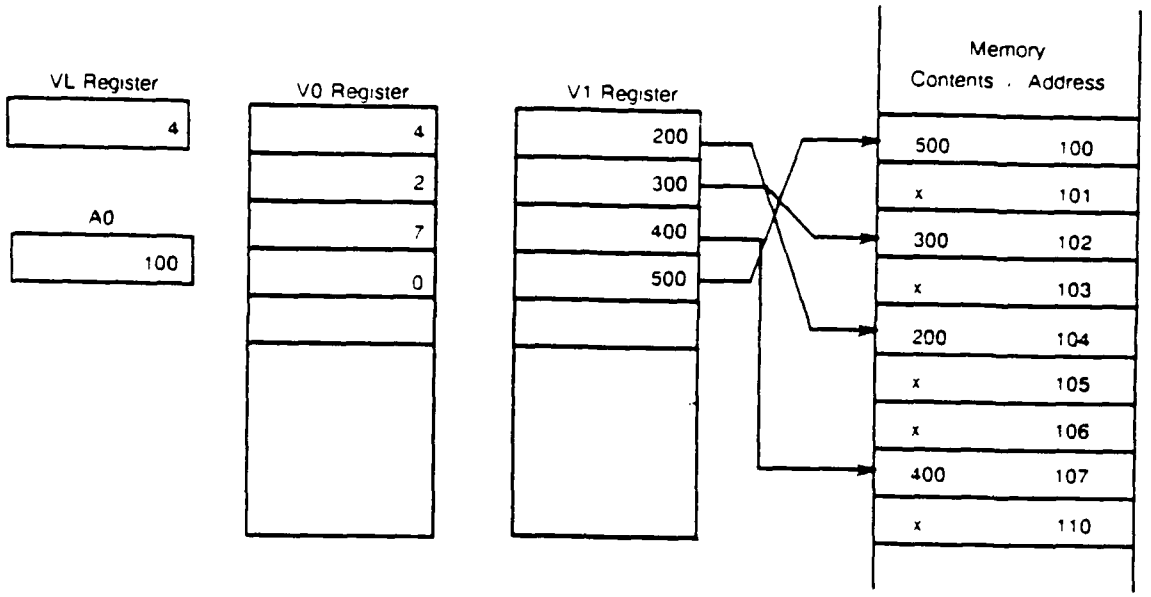


Figure 11. Cray Y-MP Scatter Command (6)

Cray Y-MP Optimization and Vectorization Resources

The Cray Y-MP optimizes and attempts to vectorize a program without any user modifications. The speed up of the WSCC's Power Stability Program resulting from compiler optimization and vectorization are presented in Chapter VII. Tools available on the Cray Y-MP to assist a programmer optimize and vectorize a program include automatic compiler optimization and vectorization, optimized library packages, and program timing utilities. Using these tools a programmer can modify a program to run faster and more efficiently on the Cray. Actual code

modification techniques are discussed in Chapter VI.

Unicos FORTRAN Compilation System

The Unicos FORTRAN Compilation System on the Cray Y-MP optimizes and vectorizes a program in three phases: a dependency analysis phase, a translation phase, and a code generation phase (15). During the dependency analysis phase the FORTRAN pre-processor analyzes the code for data dependencies and passes the information to the subsequent phases by placing directives in the code. The pre-processor also attempts to transform the code so the compiler can vectorize it during code generation (15). During the translation phase the FORTRAN mid-processor interprets the user and pre-processor directives and inserts autotasking directives. During the code generation phase the compiler performs scalar optimization and vectorization and produces relocatable object code (15).

Optimized Mathematical Libraries

The Cray Y-MP supports a wide variety of mathematical software libraries which are either totally or partially optimized. Many of the routines are written in assembly language to give maximum performance. The

mathematical libraries are continually updated to improve their performance (15).

A few mathematical libraries which are applicable to the optimization of a power system stability program are: LIBSCI, ITPACKV, and SMPAK (15).

LIBSCI. This is the default library on the Cray Y-MP It contains many vectorized and optimized mathematical and scientific routines.

ITPACKV. This library contains vectorized routines to perform iterative solutions of sparse linear equations. It is normally used for the solution of discretized partial differential equations.

SMPAK This library contains a vectorized version of the Yale Sparse Matrix Package which directly solves large sparse matrix linear equation systems.

More detailed information on libraries is available either in the on-line documentation or in Cray Research, Inc. publications.

Table 4. Summary of the Cray Y-MP Characteristics (18)

Number of processors	1,2,4,8
Peak performance	
Per processor	333 Mflops
System Total	2,670 Mflops
Cycle time	6.0 ns
Maximum main memory	1 Gbyte*
Extended memory	4 Gbyte
Technology	
Logic	2,500 gate ECL
(gate delay)	(350 ps)
Main memory	ECL
(cycle time)	(30 ns)
I/O storage: disk;solid-state secondary	
No. paths total	48;4
Max. data rate path	26.6;1,250 Mbytes/s
Max aggregate data rate	800;5000 Mbytes/s
Features	
Pipes (vector)	8 Add 8 Logical 8 Shift 8 Popcount 8 FP Multiply 8 FP Add
Load/store	16 Loads and 8 Stores
Registers, words per processor	
Scalar	64B,64T,8A,8S
Vector	512 = 8x64
Other	Vector Chaining

* In units of 2^{30} bytes =1,073,741,824, called a Gbyte.

Cray Y-MP Timing Utilities

The Cray Y-MP has five timing utilities available to help identify the CPU-intensive lines of a program.A concentrated effort should made to optimize these lines in order to get the greatest speed up.

prof. This utility gives information on the relative time spent in each subroutine and loop. It provides considerable information for minimal overhead. It is not the most precise timing tool but provides the most usable information for the least amount of additional CPU time (15).

flowtrace. This utility provides the percentage of CPU time spent in each subroutine and a calling tree for the entire program. It uses significantly more CPU time than prof, but can be invoked on limited sections of a program using compiler directives (15).

hpm (hardware performance monitor). This utility reports the machine performance during program execution. It produces four types of reports: information on the number of operations and instructions per second, the hold-issue conditions, memory use, and vectorization. It uses very little overhead but a program must be run once

for each report (15).

perftrace. This utility provides the same information as hpm, but for individual portions of the program. It requires considerable CPU time for overhead (15).

second(). This system call returns the CPU time in seconds since program execution began. It can be inserted in the program anywhere and used to time portions of the program. It uses little overhead and that overhead can be calculated and used to adjusted the total time (15).

More detailed information and examples using these tools are available either in the on-line documentation or in Cray Research, Inc. reference manuals.

CHAPTER VI

OPTIMIZATION AND VECTORIZATION TECHNIQUES

Introduction

There are numerous reasons to take the time to optimize a program. Optimization can result in a program requiring less CPU time, saving money, reducing turnaround time, and possibly receiving higher execution priority on a crowded system. A side benefit to program optimization is that system throughput is increased because system resources are more fully utilized. Additionally, solvable problems previously considered to expensive to attempt may no longer be cost prohibitive. Finally, existing programs may be expanded to incorporate more details and remove approximations (13;15).

Optimization involves arranging the data and instructions so execution takes full advantage of the computer's architecture and utilizes a minimum of its resources. According to SDSC (15), optimization can speed up program execution up to three orders of magnitude.

Vectorization on the other hand, changes scalar loops into vector operations. This is accomplished by grouping operands so they can be processed with a single vector instruction versus repeated issuance of a series of scalar instructions (15).

The Cray Y-MP compiler automatically optimizes and attempts to vectorize all FORTRAN programs during compilation (15). The programmer can assist the compiler by understanding how it attempts to optimize and vectorize the program, what inhibits optimization and vectorization, and how to eliminate these inhibitors. The first section of this chapter presents compiler techniques to optimize and vectorize a program. The second section presents programmer techniques to improve optimization and vectorization.

Unicos FORTRAN Compilation Phases

The Unicos FORTRAN Compilation System was introduced in Chapter V. Its three compilation phases are the dependency analysis phase, the translation phase, and the code generation phase. During these phases the compiler analyzes the program, optimizes and attempts to vectorize it, and generates the object file.

Dependency Analysis Phase

The Cray Y-MP FORTRAN pre-processor analyzes a program for dependencies and attempts to eliminate them by manipulating the code (15). During analysis the program's basic blocks, sections of code with single entry and exit points, are identified. Block relationships are identified and flowgraphs of these relationships are established. Then, all loops in the blocks are identified and examined for data dependencies (13). A data dependency exist if a word in memory is overwritten before it is used, or its value is used before a new value is stored there. The most common data dependency is recursion (15).

Recursive data dependencies occur when a value's calculation depends on a previously calculated value in the same vector (13). During vector processing results can only go from one register to a functional unit or to another register as discussed in Chapter V. No results are stored in memory until all elements in the vector are processed. Results cannot be individually accessed from a register, therefore the second value cannot be calculated until the first value is stored. Recursion in single dimension arrays always inhibits vectorization (12;15). If the array is multidimensional it may be possible to restructured the code so the inner loop

vectorizes. This restructuring is discussed in the section on recursive loops.

An exhaustive analysis to determine all dependencies would cause compilation time to become unacceptably long. If the pre-processor cannot determine whether a dependency exists at compilation time, it produces conditional code and then determines whether or not to vectorize the loop during program execution.

Following the dependency analysis the compiler manipulates the source code in an attempt to eliminate dependencies (13;15). Code manipulation is discussed later.

Translation Phase

The Cray Y-MP FORTRAN mid-processor examines the code produced by the pre-processor for sections which can execute concurrently. It inserts autotasking directives into the code which direct the compiler to multitask those section during execution. It also produces a listing file that can be examined (15). Programs should be tested with and without the autotasking directives. Sometimes the overhead involved in multitasking small code sections is more than the speed up gained and the program actually executes slower. This is especially true for sparse vectors where there may be very few

computations involved.

Code Generation Phase

During code generation the compiler schedules the use of functional units, assigns registers, determines whether to issue scalar or vector instructions, and schedules when instructions will be issued (13). The compiler looks for both explicit and implicit vectorization (12). FORTRAN constructs that have only vector interpretations, such as compiler directives, array syntax extensions to FORTRAN, and vectorized library calls are considered explicit vectorization (12). Implicit vectorization occurs when the compiler issues vector instructions to execute a loop (12). The Cray Y-MP compiler automatically bottomloads scalar loops when possible. In bottomloaded loops the next operands are fetched and their evaluation started before the loop index is actually tested to see if they are required. If the loop is finished, the unneeded operands are discarded and execution continues with the next instruction. Bottomloading can increase speed up by ten percent or more. Data inconsistencies rarely occur and bottomloading can be disabled if data problems do result (15).

Compiler Optimization and Vectorization

Compiler Code Manipulation

Compiler techniques used to optimize code are common subexpression and constant expression eliminations, invariant code relocation, operator strength reduction, useless code and unneeded store eliminations, expression reordering, constant-increment variable expansion, and expression replacement.

Common subexpression elimination. The compiler eliminates common subexpressions by evaluating them during loop initialization and making them register values during loop execution (12;13;15). This reduces the number of operations required during loop execution and exploits the fast register to register transfer capability of the system.

Constant expression elimination. The compiler evaluates constant expressions during compilation, stores them in memory, and replaces them with a variable in the code (12). Then, during execution the value is prefetched from memory once instead of being recalculated each time.

Invariant code relocation. Expressions which do not change during loop execution are called invariant code. They are moved out of the loop and evaluated during loop initialization (13;15). According to Levesque and Williamson (12), the programmer could move the expressions but letting the compiler move them actually increases the speed up. The compiler evaluates the expressions and assigns their values to registers during loop execution. If the programmer moves the expressions out of the loop the compiler has to store all their values in memory and then fetch them during execution. This adds unneeded memory accesses.

Operator strength reduction. Operator strength reduction replaces slower executing operators with faster ones (12;13;15). For example, division by two can be replaced with multiplication by one-half. Since multiplication is faster than division the expression is evaluated more efficiently. This type of optimization is very machine dependent, but Metcalf (13) provides the general guidelines in Table 5 on operator strength. When coding an expression a programmer should always use the faster operators when possible.

Unneeded store elimination. If a variable appears

on the left side of the assignment operator more than once during a loop, all except the last store operation are unneeded (12). Therefore, the compiler suppresses all but the last memory store and maintains the other values in registers during loop execution.

Useless code elimination. During compilation if the compiler finds code that is never used or can never be executed it will eliminate it (15). This reduces the size of the object code and makes execution more efficient by increasing both spatial and sequential locality.

Table 5. Approximate speed of arithmetic operations, fastest to slowest (13)

Integer addition and subtraction
Floating-point addition and subtraction
Integer multiplication
Floating-point multiplication
Floating-point division
Integer division
Exponentiation to a constant positive integer
Exponentiation to an integer variable
Exponentiation to a real variable

Expression reordering. The compiler examines each expression and issues instructions to evaluate it as quickly as possible. To do this it starts the evaluation of the slowest operations first so that all operations

finish at approximately the same time. This reordering may cause numerical differences in the results and test runs should be made to insure that the differences are not significant (13). If significant differences do result optimization can be turned off for that section of code using compiler directives.

Constant-increment variable expansion. A variable incremented by a constant amount at only one point during each iteration of a loop is called a constant-increment variable. The variable's range is evaluated during compilation and the variable is then treated like a vector during program execution (12). As a vector it is evaluated using the more efficient vector instructions instead of scalar instructions. An example of a constant-increment variable is the index of a DO loop.

Expression replacement. Some expression constructs appear very frequently in scientific and engineering problems, for example: first and second order linear recursions, matrix multiplications, and reduction formulas. Since they are very common, optimized routines have been developed to implement them and are available in the mathematical libraries. A programmer using compiler directives, can direct the pre-processor to

replace these constructs with optimized routines. Other constructs the pre-processor automatically tries to replace are maximum and minimum evaluations, arithmetic IF statements, and IF loops (15). To examine pre-processor modifications, the programmer can obtain a program listing which contains the modifications (15).

Compiler Loop Vectorization

Prerequisites for loop vectorization. For a loop to implicitly vectorize on the Cray Y-MP it must meet certain prerequisites. The loop must be the innermost loop, and be either a DO loop or in array syntax. The loop must perform calculations on one or more arrays or subsets of arrays, and every variable in the loop must be a scalar, vector, or index type variable (15). Additionally, all statements in the loop must be vectorizable.

Vectorizable FORTRAN statements. According to SDSC (15) the FORTRAN statements in Table 6 can be vectorized either explicitly with compiler directives or implicitly on the Cray Y-MP. According to SDSC (15) certain constructs inhibit both explicit and implicit vectorization.

Table 6. Cray Y-MP Vectorizable FORTRAN Statements (15)

Assignment statements
Conditional statements
GO TO statements that jump forward
IF(...) GO TO statements that jump forward
IF(...)THEN statements
ELSEIF(...) THEN statements
ELSE statements
ENDIF statements
Arithmetic IF statements that jump forward
CALL statements to user subroutines (requires compiler or command line directives)
FUNCTION references (if vectorized version is available)
COMMENT statements
CONTINUE statements
FORMAT statements
DATA statements

Table 7 lists statements which inhibit vectorization. Entries appearing in both Table 6 and Table 7 may or may not vectorize depending on the situation. Loops containing statements form Table 7 must be restructured before they will vectorize.

Table 7. Statements that Inhibit Vectorization (12;15)

RETURN and STOP statements
Input/Output statements
CHARACTER variables or arrays
Computed GO TO statements
Backward transfers in a loop
Certain long nested IF block
GO TO statements that exit a loop (some compilers can vectorize this statement)
CALL statements where no vectorized version is available
FUNCTIONS where no vectorized version is available
Recursions

Programmer Optimization and Vectorization Techniques

Identifying Sections of Code to Modify

Metcalf (13) argues that all programs running on or being developed for supercomputers should be checked for optimization and vectorization. He states there is no way of insuring the CPU-intensive portions of the code are running optimally without testing. Metcalf (13) presents what he calls the ten-percent ninety-percent rule, which says that ten percent of the code uses ninety percent of the time. Therefore, by thoroughly examining the code large gains in speed up can be attained by modifying only a small portion of the code.

The first step in optimizing a program is to develop a good representative test for the program (15). If a single test cannot fully simulate an actual program run, Levesque and Williamson (12) suggest running several different tests to insure that good timing statistics are available and a great deal of time is not wasted on code that is hardly used. Initially the work done on the EXCITE routine in the WSCC Power Stability Program fell into this area. The EXCITE subroutine was modified based on a short test run and its execution time was reduced by two. But during an actual stability study run the

modified program took three times longer to run than the unmodified program. Levesque and Williamson (12:84) stated it aptly, "...a good investment in time and analysis will save a lot of misguided restructuring work later."

Once a good test program has been developed, Levesque and Williamson (12) recommend finding out how much the compiler optimization and vectorization speeds up program execution. First, time the program with all optimization and vectorization turned off. Then, time the program with full optimization and vectorization. Compare the program outputs to see what changes, if any, occurred. If there are no changes or they are insignificant, compare the two run times. According to Levesque and Williamson (12) if the times are about equal then significant speed up can normally be attained. If the optimized time is five to six times faster than the original time then there may be little to gain.

Assuming the program is to be optimized, the next step is to determine what optimization and vectorization was done by the compiler, and where the most time is spent during program execution. Run the test program again using the timing utilities discussed in Chapter V and obtain program listings showing what changes were made by the compiler made (15). The listings will show

most the optimization and vectorization performed by the compiler and provide line numbers and loop markings which correlate to the timing statistics.

The timing statistics generally have two types of peaks, narrow peaks indicating that only a few lines need to be optimized or broad peaks indicating that the whole algorithm may need to be restructured. As the program is modified the narrow peaks will diminish as the law of diminishing returns is reached (13). If the loops are very long it may not be possible on the first timing analysis to identify the areas using the most time. Long loops may have to be broken into smaller loops and the timing analysis repeated. Also long loops may not vectorize because they can overtax the registers in the system (12). The program listing and timing statistics are very helpful in identifying were to break long loops. The listing marks the loops and the timing statistics pinpoint the loops or lines most often executed. Normally long loops contain fairly natural breakpoint where they can be broken, such as a line were several GO TO statements converge.

After identifying loops requiring modification, a systematic approach to restructuring should be adopted. Small portions of the code should be modified and the modification's effect on execution time and program

output should be ascertained. It is not always possible to foresee the effect of changes on a program since the compiler may optimize the code differently after modification (12-13). Therefore tests should be run after each change. If errors or data inconsistencies develop it is easier to identify their source when only small changes have been made.

SDSC (15) recommends taking the following actions to optimize a program: restructure or rewrite nonvectorizing loops, insure that the code accesses memory and functional units optimally, use fast input/output routines, and use optimized mathematical library routines where possible. Additionally, Metcalf (13) recommends not trying to do the compiler's job. Let the compiler optimize the areas described earlier, hand optimization of these areas may only inhibit compiler optimization or introduce additional delays (13).

Loop Restructuring Techniques

The compiler only vectorizes the innermost DO loop. In a vectorized loop, one vector instruction controls an operation for all the elements in a vector, provided the vector does not have more elements than the vector register. When dealing with sparse matrices the vector break-even length for a system must be known. If the

overhead required for loop vectorization is more than the time required to perform equivalent scalar calculations then the program will run slower when vectorized. Cheng (5) performed an extensive investigation of the Cray X-MP and found a vector break-even length of two elements or more. He attributed this mainly to the register to register architecture and the vector chaining in the system. Since the Cray Y-MP has these same basic characteristics its vector break-even length should parallel that of the Cray X-MP.

There are two broad classes of non-vectorizing loops: loops that do not contain data dependencies or non-recursive loops, and loops that do contain data dependencies or recursive loops.

Non-recursive loops. The first six entries in Table 7, page 83 are the most common reasons that non-recursive loops do not vectorize.

RETURN, STOP, input, and output statements always inhibit loop vectorization. On the Cray Y-MP the RETURN and STOP statements can be moved outside the loop and replaced with GO TO statements. Since the Cray Y-MP vectorizes GO TO statements which exit a loop their movement will allow vectorization (13;15). The input and output statements, if possible, should be split out of

the loop. Once outside the loop it may also be possible to replace them with an optimized routine. Input and output optimization is discussed later.

Loop splitting has been mentioned several times and is used as part of several techniques. It divides a single loop into two or more subloops which perform the same overall operation. The objective is to create one or more loops which vectorize and one loop that does not, but which contains a minimum number of operations. When splitting loops, a check must be made to see if any scalar values need to be transferred between loops. If scalars must be transferred, a technique called scalar promotion can be used. Levesque and Williamson (12) describe scalar promotion as creating a scratch array to hold the scalar values only during execution of the loops.

Any character variables or character arrays in a loop stop vectorization. It is best to avoid using character data if possible. If character data is required, try to isolate the character type operations from CPU-intensive loops (15;13).

Computed GO TO statements in loops can be rewritten with a vectorizable IF block (12;15). The computed GO TO, in actuality, performs the same sorting functions as a series of IF tests. Properly choosing the test

conditions and rearranging the code allows the compiler to optimize the loop.

Assigned GO TO statements on the other hand do not simply sort the operations. The only way to vectorize loops with assigned GO TO statements is to rewrite the loops without them (12).

Backward transfers in loops also require rewriting the loop. If the loop is not performing a convergence operation, try rewriting the loop with a forward GO TO statement or a DO loop. If the loop is for convergence, try rewriting it as a DO loop with a convergence test. When convergence is reached use a GO TO statement to exit the loop (12).

Many FORTRAN programs were written using IF tests and GO TO statements to steer execution through groups of common and flag dependent expressions. This was done to minimize program size and since all programs were scalar it had little effect on efficient execution. This programming style often results in backwards GO TO statements. These programs can be rewritten with the common expressions either incorporated into the flag-dependent sections or split out into a different loop and computed separately. A third technique to handle backwards GO TO statements is to create a new array for each flag value and then process these arrays in new

vectorized DO loops containing the appropriate expressions. If the new arrays are large the individual DO loops could be multitasked during execution.

According to Levesque and Williamson (12), very few compilers attempt to vectorize long nested IF blocks due to the low probability of ever reaching the innermost block. Their technique for dealing with long nested IF blocks requires breaking the nested blocks into a series of single level IF ELSE blocks which contain only the code applicable to the IF test. The effectiveness of this method can be very unpredictable. They present two examples where the speed up ranges from a minimal improvement to a factor of eight improvement (12).

The Cray Y-MP vectorizes GO TO statements that exit a loop. Levesque and Williamson (12) present a technique to handle this statement if the compiler cannot. It requires a combination of loop splitting and stripmining. It will not be covered here since these statements are not a problem on the Cray Y-MP.

Whether CALL and FUNCTION statements vectorize depends on their code. Calls to subprograms which do not vectorize can be handled in a number of ways. If they are non-recursive, techniques similar to those just presented can be used. If they are recursive techniques similar to those described next can be used. Additional

techniques to handle subprogram calls are presented later.

Recursive loops. Recursive loops, as discussed earlier, contain expressions that require as operands results from previous calculations in the loop. Recursion occurs in many scientific and engineering problems, especially ones using approximation routines like the Newton-Raphson method. If the compiler recognizes the recursion many times it replaces it with an optimized scalar routine and the recursion does not adversely impact on the solution time (12). The compiler may also be able to reorder the expressions to eliminate the recursion. The programmer may be able to modify the expressions so the compiler recognizes the recursion replaces them. If none of the above methods work, the compiler tries to create a temporary array to hold values during loop execution, but this is not always successful either.

Optimized recursion routines in the mathematical libraries utilize a method called loop unrolling to increase performance. Loop unrolling is a technique useful to a programmer also. Normally unrolling a loop to a depth of four is sufficient to significantly improve execution (12-13). A depth of more than four will

probably overtax the eight vector registers and require memory accesses. To unroll a loop, the loop stride is changed to the depth of unrolling. For example, a stride of one would be changed to four if a unrolled depth of four is desired. Then the expressions in the loop are copied and the indices modified so that the loop computes the same number of results each iteration as the unrolling depth (12). For example, a loop with an unrolled depth of four will compute four results each iteration instead of one result. By introducing more expressions into the loop the compiler can better utilize the functional units and registers. Loop unrolling also reduces the number of memory accesses required each iteration, adding an additional speed up factor (15).

If the compiler cannot determine whether or not a loop is recursive during compilation it will not vectorize the loop. If programmer knows that no recursion exists, a compiler directive can be used directing the compiler to ignore any ambiguous dependencies (12;15). Levesque and Williamson (12) state that anytime indirect addressing appears on both sides of an assignment operator the compiler must assume there is recursion and therefore not vectorize the loop. The Cray Y-MP compiler can vectorize expressions with indirect addressing utilizing the gather/scatter command discussed

earlier. However, direct addressing is considerably faster than indirect addressing and should be used whenever possible (15).

Nested loops. When dealing with nested loops it must be remembered that only the innermost loop can be vectorized. If the inner loop is not recursive and the order of the indices can be switched without introducing recursion, insure that the innermost loop has the largest index. Since the inner loop is the only loop vectorized this gives the processor the most elements to process each iteration (13). If the innermost loop is recursive switching loop indices may move the recursion to an outer loop allowing the innermost loop to vectorize (12). For a N dimensional array with recursion in all dimensions, Levesque and Williamson (12) suggest promoting the array dimensions to N+1 before attempting to switch the indices. This may provide an index which is not recursive and allow inner loop vectorization. Finally, examine the algorithm for possible ways to rewrite it eliminating the recursion in at least one dimension (12).

General comments on loops. Compiler listings mark vectorized and scalar loops in a program. They also list what types of optimization were performed and contain

messages pertaining to why scalar loops did not vectorize. These listings help a programmer decide where modifications need to be made to the program.

Loops containing few statements, should be combined when possible to reduce the overhead of starting numerous loops. It also gives the compiler more code to use to optimize the use of the registers and functional units (12;13). For vectorizing loops there is a point when processor resources will be overtaxed and the loop no longer vectorizes. When this happens remove expressions until the loop again vectorizes and start a second loop.

Long loops may not vectorize since there may not be enough vector registers to hold all intermediate results (12). Long loops should be split, if possible, into shorter vectorizable loops if they utilize a significant amount of execution time. By splitting long loops, it may be possible to identify small sections of code which use a large percentage of execution time. By concentrating on these sections significant speed up can be gained for minimal effort. At some point the effort required to improve performance will not be worth the gain in performance.

Memory Access Conflicts

The Cray Y-MP main memory is organized in four

sections each with eight subsections. Each subsection is divided into eight banks, for a total of two hundred and fifty-six banks. Due to memory interleaving four types of memory conflicts can occur which hinder program execution: section conflicts, subsection conflicts, simultaneous bank conflicts, and bank busy conflicts. A memory conflict occurs when the program makes a memory reference before the memory is prepared to process the request (15).

According to SDSC (15) index stride is the major determining factor of whether or not a memory conflict takes place. Index stride is defined as the number of words of memory between consecutive vector elements. An index has a unit stride when the number of words is one. Indices with non-unit strides are further classified as either even or odd strides. Vectors using indirect addressing or the gather/scatter command have a variable stride (15).

Section conflicts. Section conflicts occur when a processor attempts to access two banks in the same section simultaneously. One of the requests is held one clock period and then processed. Section conflicts happen with an index stride of four (15).

Subsection conflicts. Subsection conflicts occur when two ports of the same processor attempt to access the same subsection simultaneously. During Subsection conflicts both requests are held from one to four clock periods before they are processed. Subsection conflicts happen with index strides of eight, sixteen, or thirty-two. A stride of thirty-two causes the worst subsection delay, allowing only one word to be transferred every five clock periods (15).

Simultaneous bank conflicts. Simultaneous bank conflicts occur when two processors attempt to access the same bank concurrently. Each request is held one clock period, at which time a bank busy conflict is issued. Simultaneous bank conflicts cause minimal delay but bank busy conflicts really degrades program performance (15).

Bank busy conflicts. Bank busy conflicts are the most detrimental to program performance. They happen with index strides of sixty-four, one hundred and twenty-eight, or two hundred and fifty-six. A stride of two hundred and fifty-six references the same memory bank on every iteration (15). Table 8 contains the Cray Y-MP performance data for a simple loop using various strides. To avoid memory conflicts it is very important to check

index strides, that is why Metcalf (13) recommends expressing the index parameters as directly as possible. Using unit strides results in the best system performance, odd non-unit strides perform second best. The gather and scatter commands performances are inferior to that of odd strides but are considerably better than that of large even strides (15). If an index stride is found which degrades performance there are several remedies for the problem. For multiple dimensional arrays, try switching the array dimensions or extending them. For arrays declared next to each other in the common block pad the common block or move them apart. Extending the dimensions or padding the common block causes the elements to be skewed in memory (15).

Table 8. System Performance for Different Strides (15)

Stride	Mflops
1	105
2	88
3	81
4	59
5	69
7	60
8	40
16	20
32	10
64	7
128	5
256	3

Memory Access Optimization

Input/output operations always inhibit loop vectorization. When possible try to split the input/output statements out of CPU-intensive loops (12). Once outside the loop, it maybe possible to replace them with optimized input/output routines to speed up the transfer of data. The slowest form of input/output is formatted loop input/output. The Cray Y-MP provides two optimized routines to input and output formatted data, they are block and implied input/output. Both routines perform the input/output function one hundred and twenty-eight times faster than formatted loop input/output (15).

The fastest input/output routine is for unformatted binary data. It is one thousand nine hundred and eight times faster than formatted loop input/output and fifteen times faster than implied or block input/output (15). The major drawbacks on unformatted binary output are that it is not usable with a text editor and not readily portable to other machines. Its advantages are that it is more compact and it significantly reduces input/output time, disk usage, and CPU usage (15).

The last type of optimized input/output is buffered input/output. It has the advantage of being an asynchronous operation performed in the background,

therefore no additional CPU time is required. But it is only available for unformatted data. If there is a significant amount of formatted input/output it maybe advisable to use buffered input/output reformat the data later using a conversion program.

Subprogram Optimization

Modularity in a program is good but taken to an extreme can severely hamper optimization, vectorization, and program performance (12). Anytime a subprogram call is made, except to intrinsic or vectorized functions, the system is forced to make a context switch. The context switch requires extra time and stops the vectorization of the calling loop (12). Levesque and Williamson (12) recommend taking one of the following actions to optimize programs with subprogram calls: replace a single line functions with statement functions; pull the referenced code into the calling routine, inlining; split the call out of the loop; push the loop from the calling routine into the subprogram; or restructure the subprogram so that it vectorizes.

Replace the function call. If the function call is simple it maybe possible to replace it with a function statement. Levesque and Williamson (12) indicate that a

speed up factor of twenty was possible on a test loop using this technique.

Inline the call. Inlining the call causes the compiler to replace each call with the code of the subprogram. SDSC (15) presents advantages, disadvantages, and inhibitors to inlining. There are two main advantages associated with inlining. Inlining reduces the amount of overhead associated with the calls by eliminating context switching. It also gives the compiler a chance to better optimize the use of the scalar and vector registers and the functional units since more code is be available. There are several disadvantages associated with inlining. The size of the program may be too large and compilation may become unacceptably slow. Debugging programs with inlined code can be very difficult since the errors may appear in unexpected places. Finally, it requires recompilation of both routines every time a change is made to the called routine. The Cray Y-MP automatically tries to inline a routine provided it meets the following criteria: the routine is less than fifty lines long, the routine does not call any other routines, all common blocks between the calling and called routines are identical, the called routine is not recursive, and the called routine does not

contain any DATA, ASSIGN, SAVE, ENTRY, alternate RETURN statements or the intrinsic function LOC (15).

Split the call out of the loop. Splitting the call out of a loop allows the rest of the loop to vectorize. When a loop is split scalar variables may need to be promoted to array variables to pass values to the subprogram in the new loop (12). Metcalf (13) points out it is more efficient to pass arrays in common blocks than to pass them as parameters. When passed as parameters the processor sets up a calling table between the two programs which is not required with common block variables. Levesque and Williamson (12) present four conditions that must be met before it is safe to split a loop to isolate a subroutine call. First the call must have no side effects on the calling routine, for example it cannot change a loop parameter. Second the called routine cannot contain STOP or alternate RETURN statements. Third the dummy arguments in the called routines, if parameters are passed, which represent array values from the calling routine, cannot be array values in the called routine. Finally if the called routine calls another subprogram that subprogram must meet conditions one through three also.

Pushing the loop into the call. If a loop calls a function or routine N times during execution the compiler is required to perform N context switches, one for each loop iteration, introducing considerable overhead. If the loop can be moved into the called function or routine then only one context switch is required for loop execution (12). It maybe necessary to split a loop first to isolate the call. Then the subprogram must be rewritten to include the loop. If the subprogram is called from several places, care must be taken to insure that each time it is called the loop must be executed. If this is not the case in may not be possible to push the loop into the subprogram. Levesque and Williamson (12) present an example which achieves a speed up factor of about twenty using this technique.

Rewrite the called code. If no other way can be found to optimize the function call Metcalf (13) recommends reducing the number of calls to avoid unnecessary overhead. Levesque and Williamson (12) also recommend modifying the called subprogram so that it vectorizes. Vectorizing the subprogram reduces the time spent in the subprogram and therefore reduces the total execution time.

Additional Points to Remember

Metcalf (13) and Levesque and Williamson (12) present several other points to be aware of when programming which directly impact how the compiler optimizes a program: avoid unnecessary initialization, use DATA statements when necessary; express loop parameters directly; use weak operators and write direct equations; avoid mixed mode arithmetic or group modes so that a minimum number of conversions are required; avoid character variables; use parenthesis to group common expressions, write them the same way, and leave them in the loop; and place invariant code and constant expressions at the beginning of loops. These actions help the compiler optimally schedule the registers and functional units by removing ambiguity from the code.

The programmer can improve vectorization by restructuring loops, insuring that the memory is accessed efficiently, using optimized input/output routines, and using optimized mathematical library routines.

CHAPTER VII

POWER STABILITY PROGRAM MODIFICATIONS

Timing the Program

The WSCC's Transient Power Stability Program was installed on the Cray Y-MP to investigate the possible speed up when run on a vector supercomputer. The program run-times were provided by the computer operating system at the completion of each run. Therefore, they include no timing utility overhead. The subroutine timings were obtained using the timing utility prof. It provides the percentage of program run-time spent in each subroutine during program execution. The increment of measure provided by prof is one-hundredth of one percent, anything less is reported as zero. This percentage was then multiplied with the program run-time to obtain the subroutine run-times.

The test program performed a ten second stability study on a real power system provided by the WSCC. The first timing was done with optimization and vectorization turned off. This provided program execution time in a

scalar mode, and a set of reference results for future comparison. Then the program was compiled and run with full optimization and vectorization. This measured the effect of compiler optimization and vectorization on the unmodified program. Then each subroutine, in turn, was modified and the program re-timed to ascertain the effects of its modification on execution time. Finally, the program was compiled and executed with all subroutines modified to find the total speed up.

Original Timing Results

Table 9 contains execution times for the subprograms which were modified. Column (b) contains program times with no optimization or vectorization. Column (c) contains program times with full compiler optimization and vectorization.

Routines to modify were chosen for various reasons. The BINOP routines were chosen because they perform unformatted binary output. The function BITS was chosen because it hass few lines and a hit count, the number of times the program was found in the routine by the timing utility, ranging from seventeen to thirty-three thousand. Subroutines SOL, EXCITE, and TSCALC were chosen simce they used the largest percentages of run-time.

Table 9. Timing Results for the Power Stability Program, With and Without Compiler Optimization and Vectorization

(a) Subprogram	Run-time (sec)	
	(b) Without Compiler	(c) With Compiler
Program	562.324	231.277
BINOP1	0.675	0.139
BINOP2	2.643	0.301
BINOP3	0.056	0.023
BINOP6	1.293	1.550
BITS	12.371	8.072
EXCITE	70.797	34.715
SOL	260.806	131.851
TSCALC	67.816	9.066

Subprogram Modifications

BINOP Subroutines

The BINOP subroutines perform unformatted binary output which is later used in the program. They were modified by creating a scratch array to hold the values to be output. The BUFFER OUT command was then used to output the data in the background so that minimal additional run time was required.

Function BITS

The function BITS unpacks a variable to identify the

model being used for a particular machine. Since it was called from several subroutines throughout the program and meet the criteria for inlining, it was inlined using a compiler command line directive.

Subroutine SOL

The subroutine SOL actually solves the A.C. and D.C. network equations. It is a very complicated subroutine and contains many interdependent loops. One large loop contains many smaller convergence loops. After numerous modifications and tests it was found that the subroutine performed best when bottomloaded by the compiler. The only modification that did not increase its execution time was to replace a formatted output loop. It was replaced with a block output statement.

Subroutine EXCITE

The subroutine EXCITE solves the equations associated with the generator excitation controllers originally contained three long loops.

Excite was first modified based on a two and one-half second test program. This falsely indicated that the first loop used a larger percentage of time than it actually does. The first loop was replaced with several

sort and update loops to process controller variables by type of exciter. The time required for EXCITE during the test program was reduced to half the original time. During an actual ten second study these modifications caused total run-time to actually increase to almost three times the original time. This reaffirms the admonishment of Levesque and Williamson to insure you use a representative test.

Timing of the ten second study revealed that the first loop used about one percent of the subroutine time while the second loop used approximately ninety-seven percent. The second loop was split into four new loops. The top of the original loop contained numerous GO TO statements which converged at a line approximately half way through the loop. The first loop split was made there and several scalar temporary arrays were created to pass scalar values between the loops. The top of the second part of the loop performed two sort functions. It then executed several common expressions before jumping through the remainder of the loop to code for the different exciter types. The sorts and common expressions were put in two separate loops. The sort loop reduced the number of times the whole loop had to be executed. The common expression loop vectorized and executed more efficiently.

An attempt was made to break up the remainder of the loop by creating a loop to sort the exciters by type and consolidating the expressions for each type of exciter into individual DO loops. But, the last part of the loop contains two nested DO loops. Since the outer loop was longer, the plan was to switch the inner and outer loops since their were several exciters of each type. However it turned out that the number of times the inner loop is executed varies within similar exciter types. Since the inner loop value varied every iteration of the outer loop it was not possible to switch the loops. It was found that this portion of the loop ran fastest when it was bottomloaded by the compiler, so it was left unmodified.

Subroutine TSCALC

The subroutine TSCALC performs an variety of functions, but its most time consuming code calculates the final generator values for each iteration. It was this section that was modified. Temporary arrays were established which sorted the generators by type. The applicable code for each type of generator was then consolidated into individual loops. While the modification gives a slight improvement in speed it also produces slightly different output.

Results

Table 10 presents the execution times, speed ups, and effects on program output for all test runs. Column (b) contains the run-times with no compiler optimization or vectorization. Column (c) contains the run-times with full compiler optimization and vectorization. Column (d) contains the run-times resulting from the individual subprogram changes. Column (e) contains the speed up of full verses no compiler optimization and vectorization, it is found by dividing (b) by (c). Column (f) contains the speed up of the modifications verses no compiler actions, its found by dividing (b) by (d). Column (g) is the speed up of the modification compared to the compiler actions, it is found by dividing (c) by (d). Column (h) indicates the results of comparing the original program output to the compiler modified outputs. Column (i) indicates the results of comparing the original program output with the modified code outputs.

Speed Up from Compiler Actions

From Table 10 it can be seen that overall program speed up attained on the Cray Y-MP was 2.82. Compiler optimization and vectorization, column (e), alone produced a program speed up of 2.43. The compiler

attained the most speed up in BINOP2 and TSCALC where speed ups of 8.78 and 7.84 were reached. The speed up for the other routines averaged 2.43, with BINOP6's performance actually degrading.

Speed Up from Code Modification

The additional speed up resulting from code modifications is shown in column (g). The modifications which improved program performance the most were the buffered output routines which replaced the BINOP routines. After modification BINOP3 and BINOP6 no longer added any run-time, while BINOP2 had a speed up of 6.69, and BINOP1 a speed up of 6.32. The loop splitting done in EXCITE resulted in a speed up of 1.19 over the compiler speed up, while inlining BITS resulted in a program speed up of 1.07. The modifications made to SOL and TSCALC provided speed ups of 1.02 and 1.05. The improvement from modifying TSCALC is probably not worth the changes which resulted in the output, especially since compiler speed up resulting from bottomloading the routine was 7.48.

Program Output Changes

Column (h) shows that no program output changes

resulted form compiler optimization and vectorization. Column (i) indicates there were program output changes for the modified program and the modified TSCALC routine. Comparison of the changes from these two studies indicate that all program changes resulted from the modifications to TSCALC.

Recommendations and Conclusions

Recommendations

When modifying a program first insure the timing statistics are representative of actual program execution. Examine the program for lines where input/output operations take place. Optimizing these operations, when possible, is fairly easy and can provide significant speed up.

Examine the timing statistics for subprograms that are called a significant number of times. If they can be inlined, the reduced context switching overhead and improved resource utilization should reduce run-time. The effort required to inline a routine is minimal. In this study it required using an editor to copy the function BITS out of the code into a new file, and then using the inline option when compiling.

Table 10. Power Stability Program Run-times, Speed up and Results Comparison

Program/ Routine	Run-time (sec)			Speed up			Results	
(a)	No O/V (b)	Com O/V (c)	Mod O/V (d)	Com/ No (e) (b/c)	Mod/ No (f) (b/d)	Com/ Mod (g) (c/d)	Com/ No (h)	Mod/ No (i)
Program	562.	231.	199.	2.43	2.82	1.16	NC	D **
BINOP1	0.67	0.14	0.02	4.86	30.68	6.32	NC	NC
BINOP2	2.64	0.30	0.05	8.78	58.73	6.69	NC	NC
BINOP3	0.06	0.02	neg	2.43	und	und	NC	NC
BINOP6	1.29	1.55	neg	0.83	und	und	NC	NC
BITS *	562.	231.	215.	2.43	2.61	1.07	NC	NC
EXCITE	70.8	34.7	29.3	2.04	2.42	1.19	NC	NC
SOL	261.	132.	130.	1.98	2.01	1.02	NC	NC
TSCALC	67.8	9.07	8.65	7.48	7.84	1.05	NC	D **

* Program run-times for inlined code.
** Output differences were the same.
neg Less than 0.01%.
und Undefined due to division by zero.
D Output different
NC No ouput change

Look for long non-vectorizing loops and attempt to split them up into smaller loops which do vectorize. This must be approached carefully, if the original loop was bottomloaded and the new loops do not vectorize the increased loop overhead may slow down execution. The work done in EXCITE to create separate loops for each type of exciter demonstrated this. The long bottomloaded loop's performance was superior to individual exciter loops since they would not vectorize.

Finally, do not be to eager to make major changes to a program. It was found that most major changes attempted did not improve performance and often changed the output significantly.

Conclusions

Simulating the large interconnected power distribution systems in more detail would help system designers and controllers work more efficiently and effectively. Solving the large number of simultaneous and differential equations associated with detailed modeling often overburdens current utility company computer systems. Vector supercomputers are capable of performing the required calculations but new programs must be developed or old programs adapted.

The theory and models for stability studies were examined to understand the source of the equations that must be solved. A power stability study was examined which outlined the solution method used in a current stability program.

Supercomputer classifications were introduced and pipelining in computers was examined in detail. The Cray Y-MP architecture was presented. The resources available on the Cray Y-MP to help optimize and vectorize programs were also covered.

The compiler techniques used to optimize and vectorize a program were examined so that programming constructs could be identified which inhibit program performance. Programming techniques available to help a compiler optimize and vectorize a program were discussed.

The actions taken to optimize the WSCC,s Power Stability Program on the Cray Y-MP vector supercomputer were discussed. It was show that significant program speed up can be attained while program outputs remain unchanged.

REFERENCES

REFERENCES CITED

1. Anderson, P. M., and A. A. Fouad. Power System and Stability. Ephrata: Science Press, 1977.

2. Arrillaga, J., C. P. Arnold,and B. J. Harker. Computer Modelling of Electrical Power Systems. New York: John Wiley & Sons, Inc., 1983.

3. Betancourt, Ramon, and Antonio Gomez. "Implementation of the Fast Decoupled Load Flow on a Vector Computer." IEEE Transactions on Power Systems. Manuscript submitted for publication.

4. Byerly, Richard T., and Edward W. Kimbark, eds. Stability of Large Electric Power Systems. New York: IEEE Press, 1974.

5. Cheng, Hui. "Vector Pipelining, Chaining and Speed on the IBM 3090 and Cray X-MP." Computer, Sep. 1989: 31-46.

6. Cray Research, Inc.. CRAY Y-MP Computer Systems Functional Description Manual. Chippewa Falls, WI: Hardware Publications, 1989.

7. Glover, J. Duncan, and Mulukutla Sarma. Power System Analysis and Design. Boston: PWS, 1987.

8. Gonen, Turan. Modern Power System Analysis. New York: John Wiley & Sons, Inc., 1988.

9. Hwang, Kai, and Faye A. Briggs. Computer Architecture and Parallel Processing. New York: McGraw-Hill, 1984.

10. Kamrath, Anke. "Craynium - for FORTRAN Users: cf77 Compiles More Easily and Produces Faster Code." Gather/Scatter, Mar. 1990: 8-9.

11. Kamrath, Anke. "Craynium: UNICOS Timing Tools to Aid in Optimizing Your Codes." Gather/Scatter, Feb. 1990: 8-12.

12. Levesque, John M., and Joel W. Williamson. A Guidebook to FORTRAN on Supercomputers. San Diego: Academic Press, Inc., 1989.

13. Metcalf, Michael. FORTRAN Optimization. Orlando: Academic Press, Inc., 1985.

14. Moore, Reagan. "CRAY Y-MP and UNICOS Coming to SDSC in December." Gather/Scatter, June 1989: 1-2.

15. San Diego Supercomputer Center Documentation Group. Optimization. San Diego: Jan. 1990.

16. Sides, Stephanie. "What's Black, White, and Read All Over?" Gather/Scatter, Jan. 1990: 1-12.

17. Stevenson, William D., Jr. Elements of Power System Analysis. New York: McGraw-Hill, 1982.

18. "Supercomputer Hardware: An Update of the 1983 Report's Summary and Tables." Computer, Nov. 1989: 63-68.

19. Technical Staff. Stability Program Users' Manual. Salt Lake City: Western Systems Coordinating Council, 1986.

ABSTRACT

ABSTRACT

Detailed simulations of large interconnected power distribution systems currently cannot be performed. Vector supercomputers have the capability to solve the great number of differential and non-linear simultaneous equations associated with the models, but programs have not been developed for them. This thesis investigates the optimization of a current power stability program installed on a Cray Y-MP vector supercomputer. Theory and models for power system stability are reviewed and an outline of the program is given. A discussion of vector supercomputer pipelining and the Cray Y-MP architecture are included. Programming constructs in FORTRAN which inhibit compiler optimization and vectorization are discussed, and program modification techniques to overcome these are presented. The modifications made to the power stability program are discussed and their effects on program execution time and output are presented. It is shown that significant speed up in program execution time can be achieved with no change in program output.